# アクションゲーム アルゴリズム マニアックス

ActionGame Algorithm Maniax

松浦健一郎 ／ 司 ゆき 著

SoftBank Creative

アクションゲーム アルゴリズム マニアックス
ActionGame Algorithm Maniax
松浦健一郎／司 ゆき 著
SoftBank Creative

# アクションゲーム<br>アルゴリズム<br>マニアックス

ActionGame Algorithm Maniax

松浦健一郎／司 ゆき 著

本書に関する情報をインターネットでも公開しています。
以下のURLよりアクセスしてください。
http://www.sbpnet.jp/books/

# はじめに

本書は「アクションゲームの仕組み」を解説する本です。ゲームプログラミングの入門書としてももちろん使えますが、普通の入門書とは少し毛色が違います。本書は次のような目的にお勧めです。

・アクションゲームの中身がどんな仕組みで動いているのかを知って、ゲームをもっと楽しみたい
・アクションゲームを自分で作りたい
・学校の課題でアクションゲームを作ることになったが、いったいどこから手をつけていいのかわからない
・ダッシュやジャンプ、武器やアイテムといったいろいろな仕掛けの実現方法が知りたい
・アクションゲームの「技術カタログ」や「ネタ集」を求めている
・アクションゲームについて熱く語るための、叩き台になる本がほしい
・昔遊んだり作ったりしたアクションゲームのことを思いだしながら、懐かしい気持ちに浸りたい
・とにかくアクションゲームが好きだ

本書はいわゆる入門書ではないので、どこから読んでもかまいませんし、好きなところだけ拾い読みしても大丈夫です。図解を中心にしているので、まずは絵だけでもパラパラと、漫画を読むような気楽な気分でお読みください。一方で、各技法をシンプルにまとめたプログラムも多数掲載しているので、アクションゲームの自作にも役立てていただけることと思います。

プレイヤーの方にとってもゲームデザイナーの方にとっても、本書がアクションゲームの世界をより深く楽しむためのガイドブックになれば幸いです。

2007年初夏

松浦 健一郎 / 司 ゆき

# Contents

## Stage 00 序章 Introduction



## Stage 01 移動 Move

## Stage 02 ジャンプ Jump



## Stage 03 仕掛け Gimmick

## Stage 04 地形利用 Land Features



## Stage 05 特殊行動 Special Motion



## Stage 06 武器 Weapon

## Stage 07 アイテム Item



## Stage 08 ミッション Mission



## Appendix

## Extra Stage 攻撃 Attack



### ○サンプル

付録CD-ROMには、本書内で紹介するサンプルのプロジェクトファイルならびに実行ファイルが収録されています。付録CD-ROMの［Action¥Release］あるいは［Action¥Debug］フォルダ内にある「Action.exe」から、デモプログラムを実行することができます。

デモプログラムの実行には、「Visual C++ 2005再領布可能パッケージ（x86）」「DirectX 9.0cランタイム」「DirectXエンドユーザーランタイム（April 2007）」ならびにDirectX9に対応したビデオカードが必要になります。デモプログラムの実行方法ならびに操作方法については、13ページをご参照ください。

### ○Extra Stage

付録CD-ROMの「Extra」フォルダには、隠しステージとして、本書内で掲載しきれなかった「攻撃（Attack）」の項の解説を収録してあります。「体当たり攻撃」や「ジャンプ攻撃」「ため攻撃」「罠」といったキャラクターが行うさまざまな攻撃テクニックについて解説しています。

ダッシュやジャンプで山や谷を越え、敵をかわして目的地へ突き進む。この大胆にして繊細なゲームがアクションゲームです。ほとんどのゲームは「8方向スティック＋ジャンプボタン＋攻撃ボタン」というシンプルな操作系を利用しながらも、作品によってまったく違ったゲーム性を楽しめることが、アクションゲームの魅力だといえます。ハードウェアの進化に伴ってグラフィックやサウンドの品質は向上しましたが、「走って、跳んで、逃げて、集める」というアクションゲームの本質は昔からほとんど同じです。また最近では、携帯電話や携帯ゲーム機、あるいはFlashなどのWebゲームが普及したため、昔ながらの2Dアクションゲームが再び注目を集めています。

# 序章

# Introduction

ActionGame Algorithm Maniax

Stage

# アクションゲームの基本構成

まず最初に、アクションゲームがどのような要素から構成されているのかを確認しておきしょう。

アクションゲームに必要な要素はそう多くはありません。ほとんどの場合は、以下のような要素から構成されています（Fig. 0-1）。

### キャラクター

プレイヤーが操作するキャラクターです。人間やロボットのように人の形をしていることが多いのですが、動物や車のこともあります。

### 敵

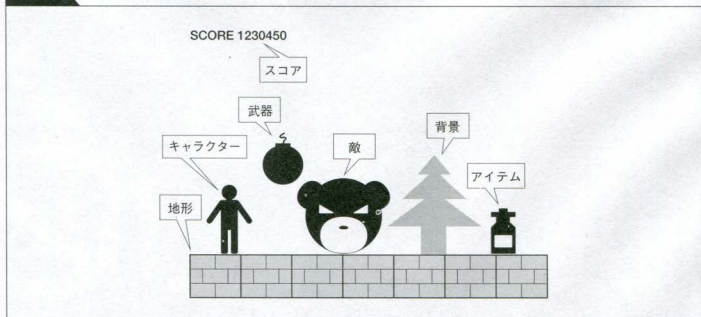
キャラクターに攻撃を仕掛けてくる相手です。多くのゲームでは、敵に触れるとミスになったり、ダメージを受けたりします。

### 武器

キャラクターが敵に向かって放つ（あるいは敵がキャラクターに向けて放つ）攻撃です。武器を敵に当てると、ダメージを与えることができます。

### 地形

ステージを構成する床や壁などです。ゲームによっては、ロープ・はしご・ベルトコンベアなど、さまざまな仕掛けが登場します。

Fig. 0-1 アクションゲームの基本構成

### 背景

キャラクターや敵の背後に表示される空や木などです。画面をにぎやかにするために表示したり、スクロールさせてスピード感を表現したりします。普通はキャラクターや敵が当たっても何も起きません。

### アイテム

キャラクターが拾うと、さまざまな効果が得られます。得点が入ったり、無敵になったりと、ゲームによっていろいろな効果が用意されています。

### スコア

得点です。多くのゲームでは、敵を倒したり、アイテムを拾ったり、ステージをクリアしたりするとスコアが入ります。

ゲームによってルールは異なります。例えば、キャラクターと敵が接触してもミスにならず、逆にスコアが追加されるというケースもあります。このように、ゲームごとに細かい差はありますが、基本的には上記のような要素の組み合わせでアクションゲームは成り立っています。

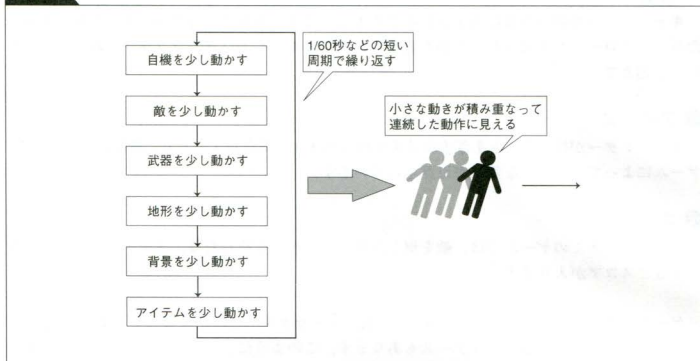
## アクションゲームの進行

アクションゲームのキャラクターは画面内を滑らかに動きますが、この動きは小さな動きの積み重ねからできています(Fig. 0-2)。例えば、画面の左端から右端までキャラクターが動く場合、最初は画面幅の1/100だけ動かし、次にまた1/100動かし…、といった動きを100回積み重ねて、連続した動きを作り出します。

この小さな動きは非常に短い周期で繰り返されます。多くのゲームでは、この周期を画面更新の周期に合わせます。例えば、家庭用TVやPC用ディスプレイの一般的な画面更新周期は1/60秒なので、ゲームも1/60秒周期で進行させます。更新の周期とゲームの周期を合わせると、アニメーションを滑らかに表示することができるからです。

敵・武器・地形・背景・アイテムなどについても、キャラクターと同じように、小さな動きを繰り返すことによって連続した動きを作り出します。アクションゲームのプログラムは、キャラクター・敵・武器・地形・背景・アイテムなどをそれぞれ小さく動かす処理を1周期として、これを何周期も繰り返します。

この「短い周期でキャラクターなどを小さく動かす処理」が、アクションゲームにおけるさまざまな動きを作り出すためのカギです。本書ではこの処理を「移動処理」と呼ぶことにします。本書の各項目では、主にこの移動処理に相当するアルゴリズムを解説し、ソースコードを掲載しています。

Fig. 0-2 アクションゲームの進行

## フレームとは

画面全体のことを指して「フレーム」と呼ぶことがあります。これはTVのような表示機器の用語ですが、ゲームでもよく使われる言葉です。

画面全体を1秒間に更新する回数のことを、「秒間30フレーム」とか「秒間60フレーム」などといいます。1/60秒周期で画面全体を更新するゲームの場合は、秒間60フレームということです。

アクションゲームをはじめとするアーケードゲームの多くは、秒間60フレームです。これはTVやディスプレイといった画面表示機器の一般的な画面更新周期に合わせてあります。

例えばキャラクターを動かすには、1フレームごとに少しずつキャラクターの座標や角度を変化させます。秒間60フレームの場合には、キャラクターを60回動かすことによって、1秒間の動きを作るというわけです。

ゲームを遊ぶときにも作るときにも、フレームという言葉は非常によく使います。例えば以下のような使い方です。

「新作の3Dゲームを買ったんだけど、30フレームでがっかりだよ」

「アクションゲームはやっぱり60フレームじゃないとなぁ！」

「なんとか頑張って1フレームに処理を押し込められない？」

なお、1秒間に更新できるフレーム数のことを「フレームレート」と呼びます。フレームレートはfps(frames per second)という単位で表記します。秒間60フレームは「60fps」ということです。ゲーマー同士の会話や、ゲームプログラマー同士の会話では、フレーム、フレームレート、fpsといった言葉を明確に区別することなく、「60フレーム」「秒間60フレーム」「フレームレー

トが60」「60fps」のようにいろいろな表現を使うことがあります。

また、「イント」という言葉もあり、フレームと同じ意味で使われます。この言葉はinterruptに由来しており、ここでinterruptは垂直同期割り込みを指しています。垂直同期割り込みは、画面更新のタイミングを知らせるもので、一般的なゲーム用のハードウェアでは1/60秒単位で発生します。

# 当たり判定処理とは

キャラクターや敵といった物体同士が接触したかどうかを判定する処理のことを「当たり判定処理」と呼びます。アクションゲームでは、キャラクターと敵が接触すると、ミスになります。キャラクターが放った武器が敵に接触したときには、敵にダメージを与えることができます。当たり判定処理によって物体同士が接触したかどうかを判定し、その結果に応じて敵を倒したり、アイテムを拾ったりするというわけです。

アクションゲームにかぎらず、あらゆるジャンルのゲームにおいて、当たり判定処理は非常に重要な要素です。シューティングゲームで自機が弾に当たったかどうかを判断するのも、格闘ゲームでパンチが命中したかどうかを判断するのも、スポーツゲームでボールを打ったかどうかを判断するのも、すべて当たり判定処理です。

先ほどキャラクターが動く仕組みについて説明しましたが、ただキャラクターが動くだけではゲームにはなりません。キャラクターを動かす処理に対して、当たり判定処理やダメージを与える処理、スコアを加算する処理などを加えることによって、初めてゲームになるのです。

## 当たり判定処理の例

当たり判定処理にはいろいろな方法があります。オーソドックスなのは、矩形を使った判定方法です(Fig. 0-3)。ゲームによっては矩形ではなく、円などを使って当たり判定処理を行う場合もあります。

なお、本書では主に2Dの当たり判定処理を扱いますが、3Dの当たり判定処理も考え方は似ています。3Dでは直方体や球を使った当たり判定処理や、ポリゴン同士の当たり判定処理を行います。

さて、本書では当たり判定処理を行うための座標情報を「当たり判定」と呼び、この情報を使用して実際に判定を行う処理を「当たり判定処理」と呼ぶことにします。Fig. 0-3では、キャラクターの当たり判定を(CL, CT)～(CR, CB)、敵の当たり判定を(EL, ET)～(ER, EB)としました。このとき、キャラクターと敵が接触する条件は次のとおりです。

```
CL<ER && EL<CR && CT<EB && ET<CB
```

キャラクターの中心座標に対して当たり判定が左右対称・上下対称の場合には、Fig. 0-4のような方法で当たり判定処理を行うこともできます。キャラクターの中心座標を(CX, CY)、敵の中心座標を(EX, EY)、X方向の当たり判定の広さをXDIST、Y方向の当たり判定の広さをYDISTとします。このとき、キャラクターと敵が接触する条件は次のとおりです。

```
abs(CX-EX)<XDIST && abs(CY-EY)<YDIST
```

absは絶対値を求める関数です。この式は、「キャラクターと敵がX方向とY方向の各々について一定距離内ならば、接触したと判定する」という意味です。

**Fig. 0-3** 当たり判定処理

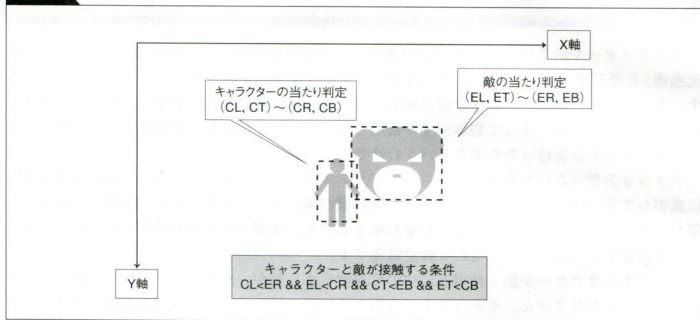
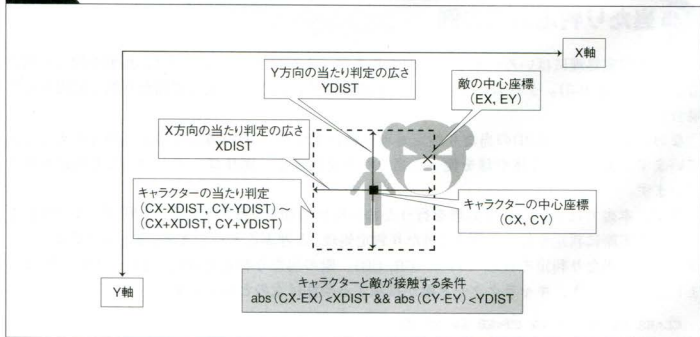



**Fig. 0-4** 左右・上下対称の場合の当たり判定処理

# タスクシステムとは

タスクシステムはゲームプログラムで使う仕掛けの1つです。ジャンルを問わずいろいろな種類のゲームに適用できますが、特にアクションゲームやシューティングゲームに向いています。ゲームによっては、タスクシステムと同じような仕組みのことを別の名前で呼んでいる場合もあります。

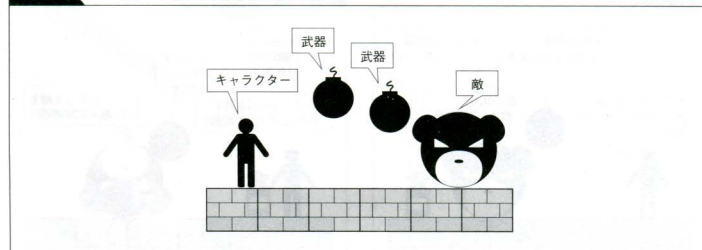
タスクシステムは複数のタスク（仕事）を並行して処理するための仕掛けです。タスクシステムを使ったゲームプログラムでは、ゲームに登場する各種の要素をタスクで表現します。例えばアクションゲームならば、キャラクター・敵・武器といった物体をそれぞれタスクとして実装します。

タスクはスレッドのようなものです。ゲームでは多くのキャラクターが並行して動きますが、タスクシステムを使うことによって、このような処理を自然な形で表現することができます。

また、タスクシステムが提供するタスクの生成や削除といった機能は、「武器を発射する処理」や「倒した敵を消す処理」などの実現に役立ちます。さらに、タスクシステムがきちんとメモリの確保や解放を管理するようにすれば、タスクの生成や削除を繰り返しても、コンピュータ上の空きメモリが断片化されて性能が落ちる心配はありません。

タスクシステムにおけるタスクの実行は、ゲームにおける物体の動きと密接に結びついています。例えば、キャラクターと敵、そして2個の武器がある状況を考えてみましょう（Fig. 0-5）。

タスクシステムを使う場合、キャラクター・敵・2個の武器を計4個のタスクで表現します。そして、これらのタスクに関して、順番に移動処理や描画処理を実行します。なお、床などの地形もタスクとして扱うことができます。

**Fig. 0-5** キャラクター・敵・2個の武器がある状況

さて、ゲームには物体を生成する機会が多くあります。例えば、

・キャラクターを出現させる
・敵を出現させる
・キャラクターが武器を発射する
・アイテムを出現させる

といった機会には、それぞれキャラクター・敵・武器・アイテムの生成処理が行われます。タスクシステムでは、こういった物体の生成をタスクの生成として扱います(Fig. 0-6)。

一方、生成とは逆に、物体を削除する機会もたくさんあります。例えば、次のような場合です。

・拾ったアイテムを消す
・敵に当たった武器を消す
・武器に当たった敵を消す
・敵に当たったキャラクターを消す
・画面外に出たアイテムを消す
・画面外に出た武器を消す
・画面外に出た敵を消す

これらはキャラクター・敵・武器・アイテムの削除処理です。タスクシステムでは、このような物体の削除をタスクの削除として扱います(Fig. 0-7)。

本書のサンプルは、タスクシステムを用いて作成されています。ただし、タスクシステムについて深く理解しなくても、まったく問題なく本書を読み進んでいただけます。タスクシステムの詳細については、拙著『シューティングゲームプログラミング』をお読みいただければ幸いです(ソフトバンク クリエイティブ刊、ISBN4-7973-3721-4)。

**Fig. 0-6** 物体の生成=タスクの生成

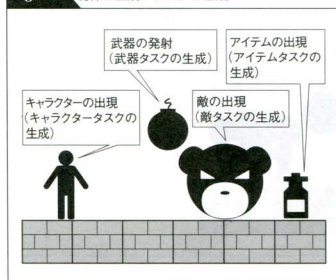
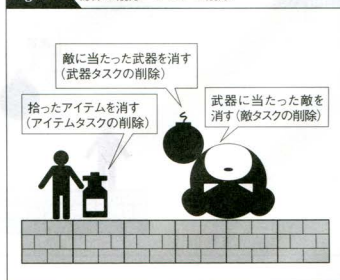
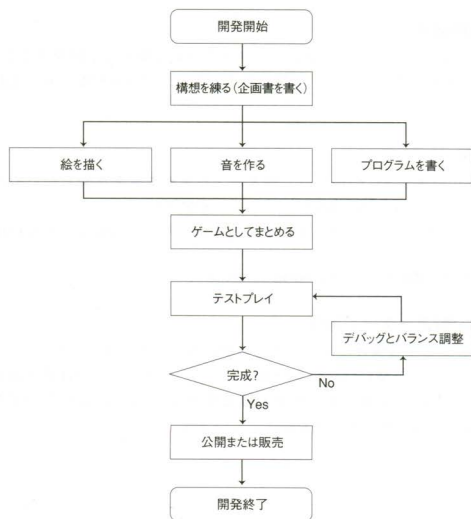



**Fig. 0-7** 物体の削除=タスクの削除

# アクションゲームを作るには

アクションゲームを作る手順は、趣味で作る場合も、仕事で作る場合も、あるいは学校の課題で作る場合も基本はそれほど変わりません(Fig. 0-8)。開発に複数の人間がかかわる場合には、いくつかの作業を並行して進めます。

開発の最初はゲームの構想を練ることから始まります。仕事の場合には企画書や仕様書を書いたり、開発予算を確保したりする必要もあります。基本的な構想がある程度できたら、絵・音・プログラムの作成に入ります。データやプログラムを作りながら、さらに構想をふくらませていくこともあるでしょう。

よいゲームを作るために大事なのはテストプレイを重ねることです。単にゲームとして動けばよいというだけではなく、時間の許すかぎりテストプレイ・デバッグ・バランス調整を繰り返します。

Fig. 0-8 アクションゲームを作る手順

# プラットフォームと開発環境

ここでは、アクションゲームのプログラムを作るために必要な準備について解説します。

## プラットフォーム

まずは、ゲームを動かすプラットフォームを選ばなくてはなりません。主なプラットフォームには次のようなものがあります。

### PC

個人がゲームを作る際に最も手軽に使えるのはPCです。選択できる言語の種類が多く、開発用のライブラリもよく整備されています。ただ、PC向けアクションゲームの市販品に関しては、アーケードゲームの移植版が低価格で発売されたりしますが、家庭用ゲーム機に比べると本数は少ないようです。

### 携帯電話/携帯端末

iモード・Palm・Pocket PC・Zaurusなどのプログラムは個人でも開発することができます。最近では端末の性能が向上したので、一昔前のアーケードゲーム程度のものも動くようになりました。

### 携帯ゲーム機

PIECE※やワンダースワンについては、個人がプログラミングを楽しむための開発キットが販売されています。ニンテンドーDSやゲームボーイアドバンスについては正式な開発キットはリリースされていませんが、エミュレータなどを用いた開発は個人でも行われています。

※PIECE
アクアプラスから発売された携帯ゲーム機（http://www.piece-me.com/）

### 家庭用ゲーム機/アーケード基板

仕事でゲームを開発する場合にはこのどちらかになることが多いでしょう。どちらもPCとは違ってハードウェアの仕様にばらつきがないので、ハードウェアの性能を限界まで生かしたゲームを作るのに適した環境です。専用の開発機材は個人ではなかなか手が出ませんが、Xbox360などは個人向けの開発キットも提供しています。

# 開発環境

趣味や同人、あるいは専門学校の課題でゲームを作る場合には、ハードウェアとしてPCを使うことが多いでしょう。PCでゲームを作る場合には、多くの開発環境のなかから好きなものを選ぶことができます。ここでは主な開発環境とその特徴を紹介します。

## C/C++とDirectX

Windowsを使う場合には最も一般的な選択です。C/C++コンパイラは各社から発売されています。DirectXとの相性のよさではMicrosoft Visual C++に軍配が上がりますが、シンプルなBorland C++ Compilerにも魅力的があります。本書で利用しているVisual C++ 2005 Express Editionは、無償にもかかわらず、デバッガなども完備したGUIベースの開発環境なのでお勧めです。

## C/C++とOpenGL

DirectXよりもOpenGLの方を好むプログラマーも少なくありません。OpenGLの利点は、DirectXほどには仕様の変化が急激ではないことと、Windows以外の環境でも利用できることです。

## Java

最近は専門学校や大学でもJavaを教えるところが多いようで、Javaは確実に普及しています。JavaはC/C++よりも速度の点では不利ですが、よいライブラリを使えばJavaでも十分な速度のゲームは作れます。

## Flash

最近ではWebブラウザさえあれば楽しめるFlashゲームが人気を集めています。FlashではActionScriptという言語を使って、ゲームなどのインタラクティブなコンテンツを作ることができます。Adobe Systems社が販売している正式な開発環境のほか、フリーの開発環境も入手できます。

## Delphi（Object Pascal）

DelphiはBorland社の開発環境です。C/C++に比べるとユーザーは少ないのですが、言語やライブラリの使い勝手がよいため、熱烈なファンを獲得している製品です。「Quadruple D」など、DelphiからDirectXを使うためのライブラリも公開されています。

## HSP

手軽に使えることで人気を集めているスクリプト言語システムです。ゲームを作りたいけれども、C/C++やDirectXを勉強するのは敷居が高いという方にもお勧めできます。

### アクションゲーム作成ツール

プログラミング言語やライブラリを勉強しなくても、絵を描いたり簡単な設定をしたりするだけでゲームが作れるツールもあります。ゲーム作成ツールといえば「ツクール」シリーズが有名どころですが、フリーのゲーム作成ツールにもかなり本格的なものがあります。

開発環境やライブラリは個人の好みで選ぶのがいちばんです。本書のプログラム例はC/C++で書きましたが、どの言語でゲームを作る場合にも基本的な考え方や手法は変わりません。本書の図や解説を参考にサンプルプログラムに手を加えれば、ほかの言語で動くプログラムを作るのも難しくないでしょう。

著者の個人的なお勧めは「C/C++とDirectX」の組み合わせか、もしくは「C/C++とOpenGL」の組み合わせです。もっと勉強が少なくてすむ組み合わせはたくさんありますが、世の中のゲームの大部分がC/C++で書かれていることを考えると、C/C++を覚えておくことにはおおいに意味があります。

特に「将来はゲームプログラマーになろう」と考えている方は、C/C++をしっかり学んでおくべきです。C/C++はゲーム以外のプログラミングにも幅広く役立つので、C/C++を身につけておけば、たぶんプログラマーとして仕事がなくなることはないでしょう。Javaも悪くはありませんが、まだまだC/C++の方が用途は広いと思われます。欲をいえば、C/C++とJavaの両方を学んでおけば完璧です。

## サンプルプログラム

本書で紹介するさまざまなアルゴリズムが実際に動いているところを確認できるように、ゲームふうのサンプルを用意し、付録CD-ROMに収録しました。実際にキャラクターを操作して、ジャンプしたり、アイテムを拾ったり、敵を倒したりすることができます。

本書はアクションゲームに登場するさまざまなアルゴリズムを紹介することを主題としているため、アクションゲームの枠組みを作る方法については解説を省略しています。しかし、付録CD-ROMに収録したサンプルは、移動・描画・キー操作など、アクションゲームの基本的な処理を備えていますので、ご安心ください。サンプルのソースコードが、アクションゲームの基本的な部分を作成する際の参考になってくれるでしょう。

また、本書内で掲載しているソースコードは、各アルゴリズムの中核となる部分を抜粋したものです。関連する処理やクラスの定義については、サンプル内の各アルゴリズムに対応する部分をご参照ください。

# サンプルプログラムの実行方法

本書の付録CD-ROMには、サンプルプログラムのソースファイル、データファイル、実行ファイルの一式が収録されています。サンプルを実行するには、「Action¥Debug」フォルダまたは「Action¥Release」フォルダの下にある「Action.exe」ファイルを、エクスプローラなどから実行してください。各サンプルの内容は、本書の対応する章で紹介しています。

サンプルプログラムを実行する場合には、事前に「Visual C++ 2005再領布可能パッケージ(x86)」「DirectX 9.0cランタイム」「DirectXエンドユーザーランタイム (April 2007)」をシステム内にインストールしておく必要があります。それぞれ、以下のWebサイトよりダウンロード可能です。

**URL** *http://www.microsoft.com/downloads/default.aspx?displaylang=ja*

サンプルプログラムの起動画面では、キーボードまたはジョイスティックで以下のような操作ができます。

・カーソルキーまたはスティックの左右でステージの選択(1ステージ単位)
・カーソルキーまたはスティックの上下でステージの選択(10ステージ単位)
・[C] キーまたはボタン2でステージの開始
・[V] キーまたはボタン3で一時停止
・[Esc] キーでプログラムの終了

サンプルプログラムには、本書で紹介する数々のアルゴリズムに対応したステージが、掲載順に並んでいます。好きなステージを選択して、開始してみてください。ステージを開始したあとは、以下のような操作ができます。

・カーソルキーまたはスティックの上下左右でキャラクターの移動
・[Z] キーまたはボタン0でジャンプや攻撃など(ステージによって異なる)
・[X] キーまたはボタン1で武器の選択など(ステージによって異なる)
・[C] キーまたはボタン2でステージ選択に戻る
・[V] キーまたはボタン3で一時停止
・[Esc] キーでプログラムの終了

サンプルプログラムの起動時オプションに「-w 1024 -h 768」などを指定すると、解像度を変更することができます。フルスクリーン時の解像度は「-fw 1280 -fh 1024」のように指定します。

## 開発環境の準備

本書では開発環境にVisual C++とDirectXを使います。付録CD-ROMに収録されたサンプルプログラムをビルドする場合には、事前に下記のソフトウェアをインストールしておく必要があります。サンプルプログラムを実行するだけならば、これらのソフトウェアは必要ありません。各ソフトウェアはマイクロソフトのダウンロードセンターなどから入手することができます。

URL **http://www.microsoft.com/downloads/**

### Visual C++

Visual C++ 2005またはVisual Studio 2005が必要です。Visual C++ 2005 Express Editionを使用する場合には、下記のURLを参考にして、Win32アプリケーションを作成するための環境設定を行ってください。また、付録CD-ROMにはVisual C++ .NET 2003またはVisual Studio .NET 2003用のプロジェクトファイルも収録しています。

URL **http://www.microsoft.com/japan/msdn/vstudio/express/visualc/usingpsdk/**

### DirectX SDK (April 2007)

本書のサンプルプログラムをビルドするためには、DirectX 9用のSDKが必要です。本書のサンプルは、2007年4月にリリースされたバージョンを使用しています。

### Platform SDK

本書のサンプルプログラムをビルドするためには、Platform SDKが必要です。「Windows Server 2003 R2 Platform SDK」をダウンロードしてインストールを行ってください。

## まとめ Stage00

本章ではアクションゲームの全体像を確認し、プログラミングの要点や開発環境の準備について整理しました。ゲームプログラミングは決して難しくはありませんが、環境を準備したり、ライブラリの性格を覚えたりと、最初はいろいろと大変なことがあります。まずは本書のサンプルを動かしたり、ビルドしたり、少し手を加えたりといったところから始めると、楽しくスムーズにアクションゲームプログラミングの世界に入っていけるでしょう。

というわけで、「アクションゲームを作るには、まずは気楽にゲームを遊ぶところから始めよう！」というのが本章のまとめです。

「移動」は、アクションゲームにおける最も基本的なアクションです。歩く、走る、ジャンプするといったあたりが移動の代表例ですが、ほかにも泳いだり氷ですべったりなど、いろいろなバリエーションがあります。

# 移動

## Move

ActionGame Algorithm Maniax

# Stage 01

## レバーダッシュ

レバーを入れっぱなしにすると、その方向にキャラクターが加速するアクションです。多くのゲームでは、ジョイスティックのレバー（あるいは移動ボタン）でプレイヤーを上下左右に移動させることができます。これだけではごく当たり前の動きなのですが、加速の要素を加えることによって、ぐっと味のある動きになります。

例えば、キャラクターが静止しているときに（Fig. 1-1）、レバーを右に入力するとキャラクターが右に進み始めます（Fig. 1-2）。さらに右にレバーを入力し続けると、キャラクターはどんどん右に加速していきます（Fig. 1-3）。キャラクターの移動があまり速くなりすぎてしまうとコントロールしにくいので、速さには上限を設けておくとよいでしょう。

レバーの入力をやめたときには、キャラクターはだんだん減速します（Fig. 1-4）。また、レバーを逆に入れたときには、急に減速させるとよいでしょう（Fig. 1-5）。

例えば、目の前に敵や罠が出現して、急に止まりたくなったときに、逆方向へのレバー入力でキャラクターが止められると便利です。この場合も、一瞬で速度を0にするのではなく、少し時間をかけて減速させるのがポイントです。こうすると、速いスピードで移動しているほど

**Fig. 1-1** 静止しているキャラクター

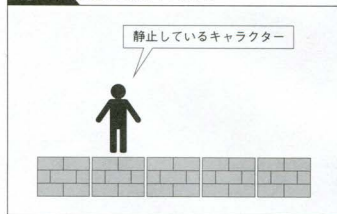



**Fig. 1-2** レバーを右に入れると右に進み始める

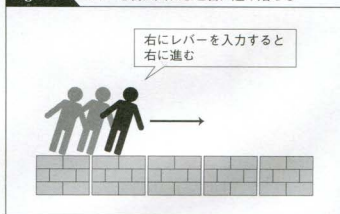



**Fig. 1-3** レバーを右に入れ続けると右に加速する

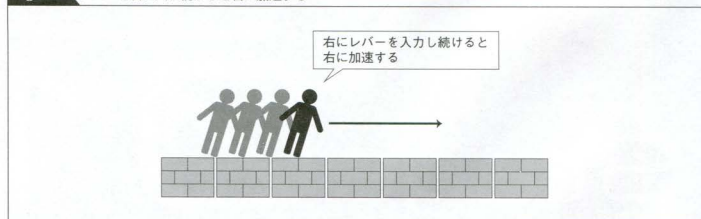



Fig. 1-4 レバーの入力をやめると減速する

Fig. 1-5 レバーを逆に入れると急に減速する

急に止まれなくなるので、リアルな動きになります。また、速く移動するほど障害物が避けにくくなるので、ゲーム性も増します。なお、急に減速したときには、土埃を立てたり、ブレーキ音を鳴らしたりといった演出があると、より楽しくなるでしょう。

レバーダッシュを使ったゲームとしては「スーパーマリオブラザーズ」や「ソニック・ザ・ヘッジホッグ」などがあります。「スーパーマリオブラザーズ」には、ボタンを使ったダッシュ（→ p. 20）も組み込まれています。「ソニック・ザ・ヘッジホッグ」には、「ループ（→ p. 53）」などを含む起伏に富んだ地形や、キャラクターが体を丸めることによってさらに加速するアクション（丸まる→ p. 294）などが盛り込まれています。

「スーパーマリオブラザーズ」と「ソニック・ザ・ヘッジホッグ」はよく似ていますが、ゲーム性はかなり違います。どちらかといえば「スーパーマリオブラザーズ」は加速やジャンプが急激なので、面のパターンをしっかり覚えて、タイミングを厳密に合わせてジャンプするゲームになっています。それに対して「ソニック・ザ・ヘッジホッグ」は、ジャンプしたあとの軌道なども調整しやすいので、もう少し気楽に遊べるゲームです。同じレバーダッシュを採用したゲームでも、加速やジャンプなどの味つけによって、このようにまったく違ったゲームになります。

レバーダッシュを使ったゲームで少し変わったものとしては「バーニンラバー」があります。これは縦スクロールのレースゲームなのですが、レバーを入れることで車の加減速ができます。

## アルゴリズム Algorithm

レバーダッシュを実現するには、レバーの入力状態に応じて、キャラクターの移動速度を変化させます。ここではキャラクターが右方向に進むことを考えましょう。

レバーが右方向に入力されていたら、キャラクターの速度を増加させます（Fig. 1-6）。キャラクターの速度をVX、加速度をaccelとすると、

```
VX+=accel
```

という計算になります。レバーが左方向に入力されたら、

```
VX-=accel
```

のように、キャラクターの速度を減少させます(Fig. 1-7)。レバーが入力されていなかったら、

```
VX-=accel*0.5f
```

のように、速度を緩やかに減少させます(Fig. 1-8)。ここではレバーを左に入れたときの半分の加速度(0.5f)で減速するようにしましたが、どのくらいの度合いで減速するのかは、もちろん自由に決めてかまいません。

**Fig. 1-6** レバーが右方向に入力されたとき

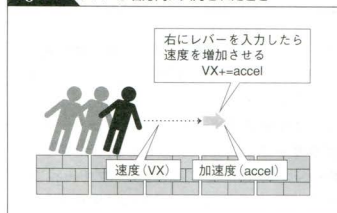



**Fig. 1-7** レバーが左方向に入力されたとき

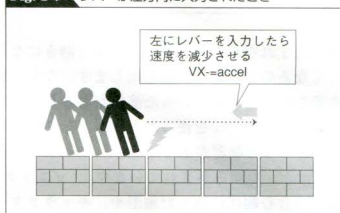
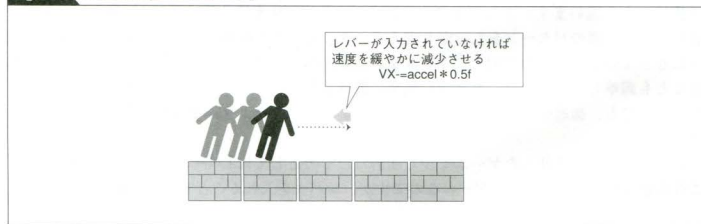



**Fig. 1-8** レバーが入力されていないとき

## プログラム Program

List 1-1はレバーダッシュのプログラムです。レバーを右に入れることで加速します。最大速度や加速度などは、自由に設定することができます。

**List 1-1** レバーダッシュ(CLeverDashManクラス)

```
// キャラクターの移動処理を行うMove関数
// このMove関数は1フレーム(1/60秒)ごとに呼び出される
// 引数のCInputStateはジョイスティックの入力状態を表す
// レバーが左に入力されているときにはis->Leftがtrueに、
// レバーが右に入力されているときにはis->Rightがtrueになる
bool Move(const CInputState* is) {

    // 最大のスピード
    float max_speed=0.5f;

    // 加速度
    float accel=0.01f;

    // 右にレバーを入力していたら右方向へ加速する
    // VXはX方向の速度
    // 背景がVXに応じてスクロールする
    if (is->Right) {
        VX+=accel;
    } else

    // 左にレバーを入力していたら減速する
    if (is->Left) {
        VX-=accel;
    } else

    // 右にも左にもレバーを入力していなかったら緩やかに減速する
    {
        VX-=accel*0.5f;
    }

    // 速度が0未満になったら0に戻す
    if (VX<0) VX=0;

    // 速度が最大スピードを超えたら最大スピードに戻す
    if (VX>max_speed) VX=max_speed;

    // 速度に応じてキャラクターを傾けて表示する
    // Angleは描画処理から参照する
    Angle=VX/max_speed*0.1f;

    // 本書のサンプルでは、Move関数がfalseを返したときに物体を消去する
    // ここではキャラクターを消去することはないので、常にtrueを返す
    return true;
}
```

SAMPLE

「LEVER DASH」はレバーダッシュのサンプルです。最初はキャラクターは静止しています。レバーを右に入力するとキャラクターが加速し、左に入力すると減速します。レバーを入力しないでいると、キャラクターは緩やかに減速します。なお、このサンプルではキャラクターを傾けて表示することで、キャラクターの速度を表現してみました。

**LEVER DASH** → p. 392

## ボタンダッシュ

ボタンを押すとキャラクターが加速するアクションです。動きはレバーダッシュに似ていますが、操作にはボタンを使います。

例えば、キャラクターが右に進んでいるときに(Fig. 1-9)、ダッシュのボタンを押すと、キャラクターが右に大きく加速します(Fig. 1-10)。このときの最大スピードは、レバーを入れているときよりも速いスピードになります。ボタンを放すと、キャラクターは元の速さまで減速します(Fig. 1-11)。

ボタンダッシュを使うことによって、キャラクターの移動速度を細やかにコントロールすることができます。速く進めるステージではダッシュのボタンを押し、慎重に進まなくてはならないステージではボタンを放すといった具合です。上手なプレイヤーには、ダッシュのボタンをずっと押しっぱなしでプレイするという楽しみもあるでしょう。

「スーパーマリオブラザーズ」では、レバーダッシュとボタンダッシュを組み合わせて使います。レバーを入れるだけでもキャラクターが加速しますが、ボタンを押すとさらに大きく加速します。また、ボタンを押しているときの方が、最大スピードもアップします。

レバーダッシュとボタンダッシュを組み合わせることによって、キャラクターが移動するときに常に加速度がかかった状態になります。これはキャラクターの動きを滑らかに見せるとともに、慣性がついた独特の操作感覚を生み出す効果があります。

**Fig. 1-9** 右に進んでいるキャラクター

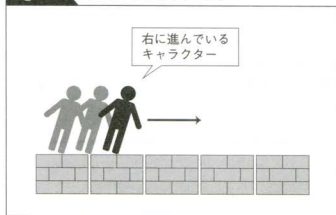



**Fig. 1-10** ボタンを押すと右に大きく加速する

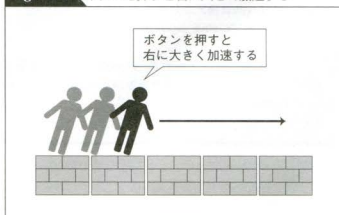
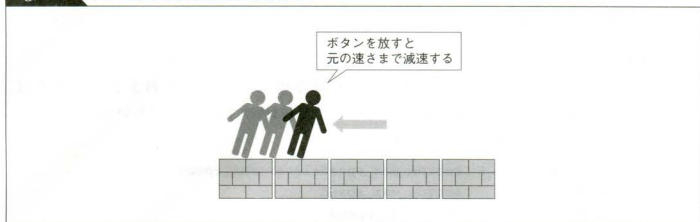



**Fig. 1-11** ボタンを放すと元の速さまで減速する

## アルゴリズム Algorithm

ボタンダッシュを実現するには、ボタンの入力状態に応じて、キャラクターの速度に対して加速度を加えます。ここではキャラクターが右方向に進むことを考えましょう。

ボタンが押されていたら、キャラクターの速度を増加させます(Fig. 1-12)。キャラクターの速度をButtonVX、加速度をbutton_accelとすると、

```
ButtonVX+=button_accel
```

という計算になります。ボタンが放されたら、

```
ButtonVX-=button_accel
```

のように、キャラクターの速度を減少させます(Fig. 1-13)。ここでは加速にも減速にも同じ大きさの加速度を使っていますが、それぞれで大きさを変えてもよいでしょう。例えば、加速しやすいけれども減速しにくいとか、逆に減速しやすいけれども加速しにくい、といった特性のキャラクターも面白そうです。

ボタンダッシュとレバーダッシュを組み合わせる場合には、ボタンダッシュの速度とは別に、レバーダッシュ(→ p .16)の速度を求めます。そして、ボタンダッシュの速度とレバーダッシュの速度を加えたものを、キャラクターの速度とします(Fig. 1-14)。前者をButtonVX、後者

**Fig. 1-12** ボタンを押したとき

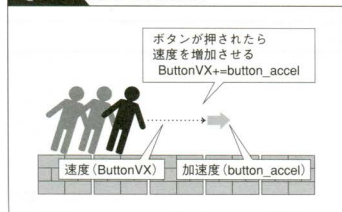
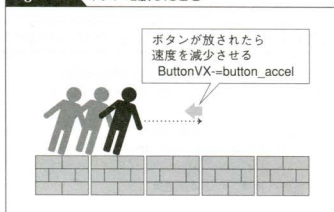



**Fig. 1-13** ボタンを放したとき

をLeverVX、キャラクターの速度をVXとすると、

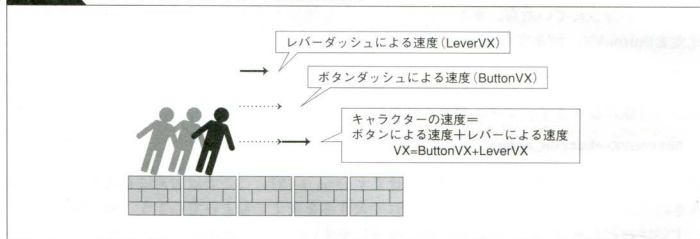
```
VX=ButtonVX+LeverVX
```

となります。

さらに、ButtonVXとLeverVXにはそれぞれ速さの上限を設けておきます。例えば、ButtonVXの上限はbutton_max_speed、LeverVXの上限はlever_max_speedといった具合です。こうするとキャラクターの最大スピードは、

・ボタン+レバーの最大スピード：button_max_speed+lever_max_speed
・ボタンのみの最大スピード　　：button_max_speed
・レバーのみの最大スピード　　：lever_max_speed

となって、ボタンとレバーを組み合わせたときに最も速いスピードが出るようになります。好みに応じて、「スーパーマリオブラザーズ」のようにボタンのみではキャラクターが移動しないようにすることも可能です。

**Fig. 1-14** ボタンダッシュとレバーダッシュを組み合わせる

## プログラム　Program

List 1-2はボタンダッシュのプログラムです。ボタンを押している間は加速を続けます。移動速度や加速・減速度を調節してみてください。

**List 1-2** ボタンダッシュ(CButtonDashManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // ボタンのみの最大スピード
    float button_max_speed=0.3f;

    // ボタンによる加速度
    float button_accel=0.006f;
```

```
// レバーのみの最大スピード
float lever_max_speed=0.2f;

// レバーによる加速度
float lever_accel=0.004f;

// 右にレバーを入力していたら右方向へ加速する
if (is->Right) {
    LeverVX+=lever_accel;
} else

// 左にレバーを入力していたら減速する
if (is->Left) {
    LeverVX-=lever_accel;
} else

// 右にも左にもレバーを入力していなかったら緩やかに減速する
{
    LeverVX-=lever_accel*0.5f;
}

// レバーによる速度が0未満になったら0に戻す
if (LeverVX<0) LeverVX=0;

// レバーによる速度が最大スピードを超えたら最大スピードに戻す
if (LeverVX>lever_max_speed) LeverVX=lever_max_speed;

// ボタンを押していたら加速する
// is->Button[ボタン番号]はボタンの入力状態を示す
// ボタン0が入力されているときにはis->Button[0]がtrueになる
// ボタン0がジョイスティックのどのボタンに対応するのかは、
// Windowsのコントロールパネルで設定することができる
if (is->Button[0]) {
    ButtonVX+=button_accel;
} else

// ボタンを放していたら減速する
{
    ButtonVX-=button_accel;
}

// ボタンによる速度が0未満になったら0に戻す
if (ButtonVX<0) ButtonVX=0;

// ボタンによる速度が最大スピードを超えたら最大スピードに戻す
if (ButtonVX>button_max_speed) ButtonVX=button_max_speed;

// レバーによる速度とボタンによる速度を加算したものをキャラクターの速度とする
VX=LeverVX+ButtonVX;
```

**List 1-2**

```
    // 速度に応じてキャラクターを傾けて表示する
    Angle=VX/(lever_max_speed+button_max_speed)*0.1f;

    return true;
}
```

**SAMPLE**

「BUTTON DASH」はボタンダッシュのサンプルです。最初はキャラクターは静止しています。ボタンを押すとキャラクターが加速し、ボタンを放すと減速します。また、レバーダッシュのサンプルと同じように、レバーを使ってキャラクターを動かすこともできます。ボタンとレバーを組み合わせると、より速いスピードで移動することが可能です。

**BUTTON DASH** → p. 392

## レバー2段ダッシュ

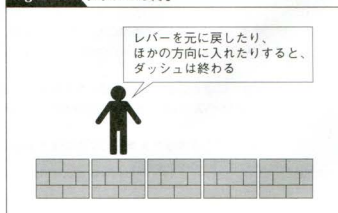
レバーを同じ方向に2回入力すると、キャラクターが速いスピードで移動するアクションです。レバーダッシュやボタンダッシュではキャラクターが緩やかに加速しますが、レバー2段ダッシュではキャラクターが一気に加速します。

例えば、キャラクターが静止しているときに、レバーを右に2回素早く入力すると、キャラクターが右にダッシュします(Fig. 1-15)。レバーを右に入れているかぎりダッシュは続きます。レバーを元に戻したり、ほかの方向に入れたりするとダッシュは終わります(Fig. 1-16)。

レバー2段ダッシュを行うには、レバーを同じ方向に2回素早く入れる必要があります。レバーを1回しか入れなかったり、ゆっくり2回入れたりしたときにはダッシュは始まりません。

レバーを同じ方向に2回素早く入れるという操作は、一種のコマンド入力です。こういった入力方法は、格闘要素のあるゲームによく似合います。レバーダッシュやボタンダッシュとは違って、一気にキャラクターを加速させることができるため、とっさに回避行動をさせたり、

**Fig. 1-15** ダッシュの開始

**Fig. 1-16** ダッシュの終了

相手を急に攻撃したりするゲームとの相性がぴったりです。

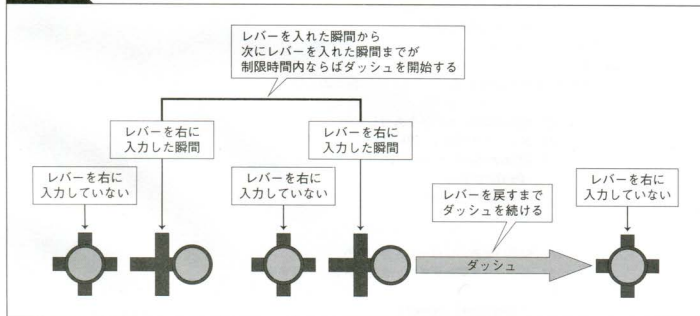
「ゴールデンアックス」はレバー2段ダッシュを採用したゲームです。レバー2段ダッシュでキャラクターをダッシュさせると、敵の攻撃をかわしたり、素早く敵に近づいたりすることができます。また、ダッシュ中にジャンプすると普通よりも高く飛べたり、ダッシュ中には特別な攻撃が出たりします。

## アルゴリズム　Algorithm

レバー2段ダッシュを実現するには、レバーを入れた瞬間から、次にレバーを入れた瞬間までの時間を調べます（Fig. 1-17）。この時間が一定の制限時間内ならば、ダッシュを開始します。

制限時間を短くすると、レバーを素早く2回入れる必要があります。制限時間を長めにすると、レバーをゆっくり2回入れてもダッシュを出すことができます。ただし、あまり制限時間を長くすると、普通に移動しようと思ったときにも誤ってダッシュが出てしまうことが多くなります。制限時間は適度に短い方が、快適に遊ぶことができます。ゲームの内容によりますが、0.2秒や0.3秒といった時間にするとよいでしょう。

レバーを入れた瞬間かどうかを判定するには、直前のレバーの状態を保存しておき、現在のレバーの状態と比べます。直前にレバーを入力しないで、現在レバーを入力していたら、レバーを入れた瞬間だということです。

Fig. 1-17　レバー2段ダッシュの実現

## プログラム　Program

List 1-3はレバー2段ダッシュのプログラムです。レバー入力の制限時間は約1/3秒（20フレーム）に設定してあります。

**List 1-3** レバー2段ダッシュ(CDoubleLeverDashManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // ダッシュ時のスピード
    float dash_speed=0.6f;

    // 通常時のスピード
    float normal_speed=0.2f;

    // 2回目のレバーを入れるまでの制限時間(フレーム数)
    // 1フレームが約1/60秒なので、20フレームは約1/3秒
    // 約1/3秒以内に2回のレバーを入力すると、ダッシュが始まる
    int dash_time=20;

    // ダッシュ中の処理
    // 右にレバーを入れるのをやめたら、ダッシュを中止し、速度を0にする
    if (Dash) {
        if (!is->Right) {
            Dash=false;
            VX=0;
        }
    } else

    // ダッシュしていないときの処理
    {
        // 右にレバーを入れた瞬間の処理
        // PrevRightは直前のフレームにおけるレバーの状態
        if (is->Right) {
            if (!PrevRight) {

                // 制限時間内にレバーを入力したら、
                // ダッシュを開始し、速いスピードにする
                if (Time<dash_time) {
                    Dash=true;
                    VX=dash_speed;
                } else

                // 制限時間外にレバーを入力したら、
                // ダッシュは開始せずに、通常のスピードにする
                {
                    VX=normal_speed;
                }

                // タイマーを0にする
                // このタイマーは、入力制限時間内かどうかを判定するために使う
                Time=0;
            }
        } else
```

```
        // レバーを入れていないときの処理
        // 速度を0にする
        {
            VX=0;
        }
    }

    // タイマーを加算する
    Time++;

    // レバーの状態を保存しておく
    PrevRight=is->Right;

    // 速度に応じてキャラクターを傾けて表示する
    Angle=VX/dash_speed*0.1f;
    return true;
}
```

SAMPLE

「DOUBLE LEVER DASH」はレバー2段ダッシュのサンプルです。最初はキャラクターは静止しています。レバーを素早く右に2回入力するとキャラクターがダッシュし、レバーを戻すと静止します。レバーを右に1回だけ入力したり、2回ゆっくり入力したときには、キャラクターは遅いスピードで右に移動します。ダッシュが始まるのは、レバーを約1/3秒以内に2回入力したときです。

**DOUBLE LEVER DASH** → p. 392

# 連打ダッシュ

ボタンを連打することによって、キャラクターが加速するアクションです。連打のスピードが速いほど、キャラクターのスピードも速くなります。

例えば、キャラクターが静止しているときにボタンを連打すると、キャラクターが動き出します(Fig. 1-18)。ボタンを速く連打するほどキャラクターが加速します(Fig. 1-19)。ボタンの連打をやめたり、連打の速さを緩めると、キャラクターはだんだん減速していきます(Fig. 1-20)。

連打ダッシュを全面的に取り入れたゲームが「ハイパーオリンピック」です。このゲームでは、連打ダッシュでキャラクターを速く走らせたり、あるいはジャンプと組み合わせて遠くに飛ばせたりします。ゲームの題材もスポーツですが、プレイヤーも常に高速でボタンを連打し続ける必要があるので、ある意味ではゲームをプレイすること自体がスポーツだともいえます。昔のゲームセンターでは、爪でボタンをこすって連打したり、腕をけいれんさせて連打したり、はたまた定規をはじいたときの振動を使ってボタンを連打したり(これは一部では禁じ手とさ

れていました）といった光景が見られたものです。

「パックランド」でも連打ダッシュが採用されています。こちらの連打の激しさは「ハイパーオリンピック」ほどではありませんが、襲ってくるモンスターをかわしたり、ジャンプ台でタイミングよく踏み切って池を跳び越えたりと、連打ダッシュとジャンプを組み合わせた歯ごたえのあるアクションが楽しめます。

**Fig. 1-18** ボタンの連打でダッシュを開始

**Fig. 1-19** 連打が速いほどスピードも速くなる

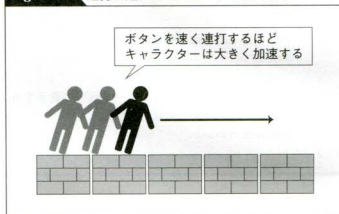
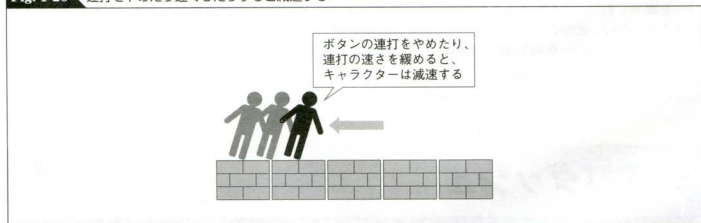
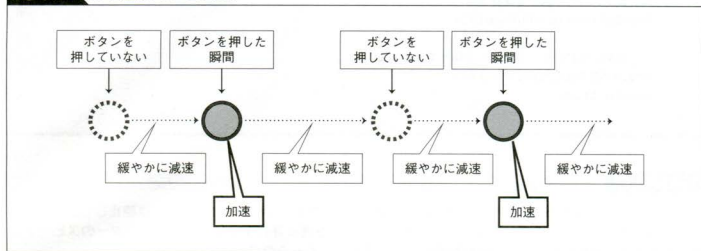



**Fig. 1-20** 連打をやめたり遅くしたりすると減速する

## アルゴリズム Algorithm

ボタン連打ダッシュを実現するには、ボタンを押した瞬間にキャラクターを加速させます。そして、ボタンを押した瞬間以外にはキャラクターを緩やかに減速させます（Fig. 1-21）。こうすると、ボタンを速く連打するほどキャラクターが頻繁に加速することになり、スピードが速くなります。

加速と減速の度合いを変えると、加速のしやすさが変わります。加速に比べて減速は弱めに、例えば減速は加速の1/10の度合いといった具合に抑えておかないと、非常に加速がしにくくなってしまいます。ここはゲームの性質に応じて、ちょうどよいバランスを探す必要があります。

Fig. 1-21 ボタン連打ダッシュの実現

## プログラム Program

List 1-4はボタン連打ダッシュのプログラムです。ボタンを連打することで、加速しながら前進します。ボタンを放すとゆっくりと減速して、最後には停止します。

**List 1-4** 連打ダッシュ(CRapidButtonDashManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 最大スピード
    float max_speed=0.6f;

    // 加速度
    float accel=0.03f;

    // ボタンを押した瞬間は加速する
    if (!PrevButton && is->Button[0]) {
        VX+=accel;
    } else

    // ボタンを押した瞬間以外は緩やかに減速する
    {
        VX-=accel*0.1f;
    }

    // 速度が0未満になったら0に戻す
    if (VX<0) VX=0;

    // 速度が最大スピードを超えたら最大スピードに戻す
    if (VX>max_speed) VX=max_speed;
```

**List 1-4**

```
        // ボタン入力の状態を保存する
        PrevButton=is->Button[0];

        // 速度に応じてキャラクターを傾けて表示する
        Angle=VX/max_speed*0.1f;
        return true;
}
```

**SAMPLE**

「RAPID BUTTON DASH」はボタン連打ダッシュのサンプルです。最初はキャラクターは静止しています。ボタンを連打するとキャラクターがダッシュします。ボタンを速く連打するほど、キャラクターのスピードは速くなります。連打をやめたり、連打を遅くしたりすると、キャラクターは減速します。

**RAPID BUTTON DASH** → p. 392

# スピードアップアイテム

アイテムを拾うことによって、キャラクターの移動スピードが上がるアクションです。アイテムには有効時間があるため、一定時間を過ぎると効果が切れて元のスピードに戻ります。

スピードアップの効果は、アイテムを拾うことによって発動します(Fig. 1-22)。アイテムを拾うと、キャラクターの移動スピードがアップします(Fig. 1-23)。アイテムには数秒や数十秒といった有効時間があります。この有効時間が過ぎると、アイテムの効果が切れて、キャラクターの移動スピードは元に戻ります。

スピードアップアイテムは多くのゲームに取り入れられています。例えば「スーパーマリオブラザーズ」では、星のアイテムを拾うと移動スピードが速くなり、さらにキャラクターが無敵になります。この効果は一定時間続きます。アイテムが効いている間はキャラクターが光り、

**Fig. 1-22** スピードアップアイテムを拾う

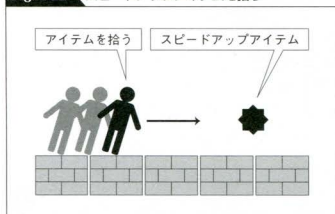
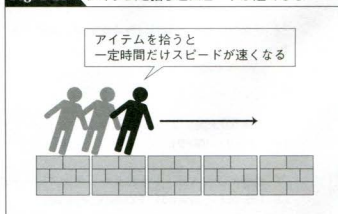



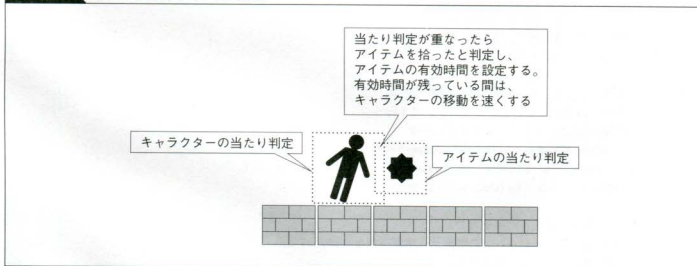
**Fig. 1-23** アイテムを拾うとスピードが速くなる

音楽も変わるので、アイテムが有効なのかどうかがすぐにわかります。このように、アイテムが効いている間は特別なエフェクトを表示するとか、特別な音を出すといった工夫があると、ゲームが遊びやすくなるでしょう。

## アルゴリズム Algorithm

スピードアップアイテムを実現するには、まずアイテムを拾ったかどうかを判定します(Fig. 1-24)。これはキャラクターと画面上にあるすべてのアイテムの間で当たり判定処理を行い、キャラクターとアイテムが接触したら、アイテムを拾ったことにします。キャラクターとアイテムが完全に重なったときだけではなく、ある程度以内の距離に近づいたら接触したことにすると、アイテムが拾いやすくなります。

アイテムを拾ったら、アイテムの有効時間をセットします。そして、時間とともに残り有効時間を減らしていきます。有効時間がなくなったらアイテムの効果は終わりです。アイテムが有効な間はキャラクターの移動を速くし、無効になったら通常のスピードに戻します。

スピードアップアイテムの有効時間を決めるときには、アイテムの出現頻度とのバランスを考えることが大事です。有効時間が長いときには出現頻度を低めに、有効時間が短いときには出現頻度を高めにするとよいでしょう。

**Fig. 1-24** スピードアップアイテムの実現



## プログラム Program

List 1-5はスピードアップアイテムのプログラムです。ここでは、アイテムの有効時間を約1秒(60フレーム)に設定しています。お好みに合わせて有効時間をいろいろと調整してみてください。

**List 1-5** スピードアップアイテム(CSpeedUpItemManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // アイテム使用時のスピード
    float item_speed=0.6f;

    // 通常時のスピード
    float normal_speed=0.2f;

    // アイテムの有効時間(フレーム数)
    float item_time=60;

    // アイテムが有効ならば、アイテム使用時のスピードで移動する
    // また、アイテムの残り有効時間を減らす
    if (ItemTime>0) {
        VX=item_speed;
        ItemTime--;
    } else

    // アイテムが無効ならば、通常時のスピードで移動する
    {
        VX=normal_speed;
    }

    // 右にレバーを入れていないときには速度を0にする
    if (!is->Right) VX=0;

    // アイテムを拾ったかどうかを判定するため、
    // すべての物体について当たり判定処理を行う
    // CTaskIterはタスクに対するイテレータ
    // (LibGame¥Task.h, LibGame¥Task.cpp参照)
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();

        // 物体の種類がアイテムで、
        // かつキャラクターとの距離が一定値以内ならば、
        // アイテムを拾ったとする
        // アイテム有効時間を設定し、アイテムを消去する
        if (
            mover->Type==1 &&
            abs(mover->X-X)<1.0f &&
            abs(mover->Y-Y)<1.0f
        ) {
            ItemTime=item_time;
            i.Remove();
        }
    }
```

```
    // 速度に応じてキャラクターを傾けて表示する
    Angle=VX/item_speed*0.1f;
    return true;
  }
```

「SPEED UP ITEM」はスピードアップアイテムのサンプルです。右にレバーを入れると、キャラクターが右に移動します。ときどきアイテムが流れてきます。アイテムを拾うと、キャラクターのスピードが通常時よりも速くなります。一定時間(約1秒)が過ぎると、アイテムの効果が切れて通常時のスピードに戻ります。

**SPEED UP ITEM** → p. 392

## 氷ですべる

氷の上で急に停止や方向転換ができずに、すべってしまうアクションです。すべっている間はキャラクターが操作不能になるため、敵の攻撃に当たったり、罠にひっかかったりしやすくなります。

キャラクターが右に進んでいるとしましょう(Fig. 1-25)。レバーを右に入れている間は普通に進んでいますが、レバーの入力をやめて止まろうとすると、すぐには止まれずにすべり出します(Fig. 1-26)。すべっている間、キャラクターは一時的に操作不能になります。しばらくすべると、ようやく止まります。

同じように、キャラクターが右に進んでいるときに、レバーを左に入れて方向転換しようとすると、すぐには方向転換できずにすべり出します(Fig. 1-27)。すべっている間はやはり操作不能です。しばらくすべると、ようやく方向転換します(Fig. 1-28)。

**Fig. 1-25** 右に進んでいるキャラクター

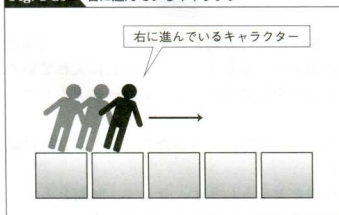
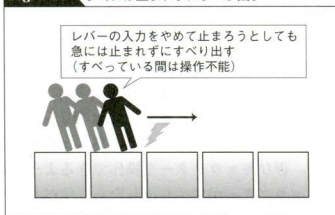



**Fig. 1-26** すぐには止まれずにすべり出す

たいていのゲームでは、氷が出てくるシーンや床が凍っているシーンでは、キャラクターがすべってしまって普通の地面にいるときよりも操作が難しくなります。例えば「アイスクライマー」は、氷の山を登っていくゲームだけあって、すべってしまう氷や凍った床が数多く出てきます。急停止や急な方向転換ができないため、慎重なプレイが要求されます。

氷でキャラクターがすべったときには、いかにもすべっているようなグラフィックを表示するとよいでしょう。見た目にも楽しいですし、キャラクターがすべっていることがよくわかるので、プレイもしやすくなります。

また、氷や凍っている床を出現させるときには、通常の地面や床との区別がはっきりわかるようなグラフィックにするべきです。すべる部分かどうかがわかりやすければ、すべらない部分を狙って止まったり方向転換したりといったプレイも可能になります。

ゲームによっては氷だけではなく、水に濡れた床や、オイルがまかれた床が登場することもあります。こういった床も氷と同様に、急な停止や方向転換ができなくなっています。

また、氷の上では停止や方向転換がしにくいだけではなく、歩き出すときにもすべるようにして、さらに操作を難しくしているゲームもあります。一方で、「アイスクライマー」のように停止や方向転換のときだけすべるようにしても、氷の雰囲気は十分に演出することができます。

**Fig. 1-27** すぐには方向転換できずにすべり出す

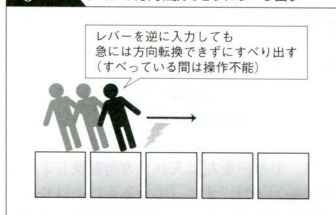
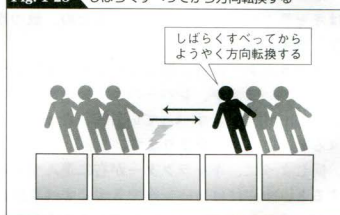



**Fig. 1-28** しばらくすべってから方向転換する

## アルゴリズム Algorithm

氷ですべるアクションを実現するには、キャラクターが進む方向とレバーを入力した方向との関係に注目します。例えば、キャラクターが右に進んでいるときにレバーを右に入れていたら、そのまま右に進みます（Fig. 1-29）。しかし、レバーを左に入れたり、レバーの入力をやめたりしたらすべり出します（Fig. 1-30）。

キャラクターがすべっている間は操作不能にします。一定時間すべったら、通常の状態に戻して、再びキャラクターを操作できるようにします。

キャラクターがすべり出したら、すべりの有効時間をセットします。そして、時間とともに残りの有効時間を減らしていき、有効時間がなくなったらすべりを終了します。すべっている間はキャラクターを一定の速度で移動させ、レバーの入力は無視します。すべりが終わったら、通常どおりレバーの入力にしたがってキャラクターを移動させます。

すべりの有効時間を長くすると、よくすべる氷になります。逆に短くすると、あまりすべらない氷になります。よくすべる氷と、あまりすべらない氷を交ぜても面白いでしょう。ただし、両者をはっきり区別できるように、グラフィックを変えるなどの工夫も必要です。

**Fig. 1-29** すべらない場合

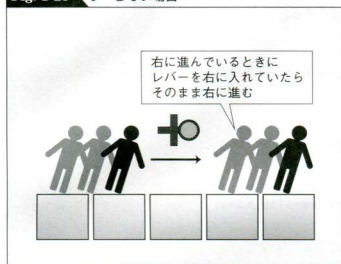
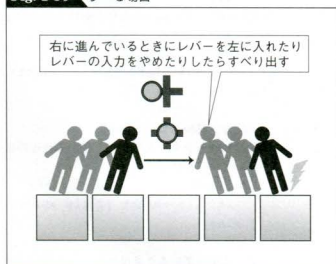



**Fig. 1-30** すべる場合

## プログラム Program

List 1-6は氷ですべるアクションのプログラムです。キャラクターの移動とは逆の方向にレバーを入れると、氷ですべって一定時間操作を受け付けなくなります。すべる時間を増やせば、よくすべる氷のように感じさせることができます。

**List 1-6** 氷ですべる(CSlipOnIceManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.1f;

    // 氷ですべる時間
    int slip_time=30;

    // 氷ですべっているとき
    if (SlipTime>0) {

        // すべる残り時間を減らす
        SlipTime--;

        // 残り時間が0になったら、いったん速度を0にする
        if (SlipTime==0) VX=0;
```

**List 1-6**

```
} else

// すべっていないとき
{
    // 右に移動しているときに、
    // レバーを左に入れたり、右に入れるのをやめたりすると、
    // すべり始める
    if (VX>0) {
        if (is->Left || !is->Right) SlipTime=slip_time;
    } else

    // 左に移動しているときに、
    // レバーを右に入れたり、左に入れるのをやめたりすると、
    // すべり始める
    if (VX<0) {
        if (is->Right || !is->Left) SlipTime=slip_time;
    } else

    // 止まっているときには、
    // レバーを入れた方向へ移動を開始する
    {
        if (is->Left) VX=-speed;
        if (is->Right) VX=speed;
    }
}

// X座標を更新してから、
// 画面端からはみ出さないように補正する
X+=VX;
if (X<0) X=0;
if (X>MAX_X-1) X=MAX_X-1;

// 移動中であることを表現するために、
// キャラクターを移動方向に傾けて表示する
// すべっている間はキャラクターを左右に振動させて、
// 足がすべってバタバタしている様子を表現する
if (SlipTime>0) Angle=-Angle; else Angle=VX*0.5f;

return true;
}
```

SAMPLE

「SLIP ON ICE」は氷ですべるアクションのサンプルです。レバーを入れた方向にキャラクターが移動します。レバーを放すか反対方向に入れるとキャラクターがすべり始め、一定の期間（距離）をすべると停止します。すべっている間は、レバーによる操作を受け付けなくなります。

**SLIP ON ICE** → p. 392

# 泳ぐ

水のなかで泳ぐアクションです。陸上とは違って、水中ではキャラクターが浮かぶので、上下左右に泳ぎまわって移動することができます。ただし、常に泳いでいないと沈んでしまいます。操作方法が地上とは異なるので、障害物や敵を上手に避けるには慣れが必要です。

水中での左右の動きについては、地上にいるときと同じです(Fig. 1-31)。レバーを右に入れると右に進み、左に入れると左に進みます。

独特なのは上下の動きです。水中でボタンを押すと、キャラクターは浮かびます(Fig. 1-32)。しかし、ボタンを押さなかったり、押しっぱなしにしていると、キャラクターは少しずつ沈んでいきます(Fig. 1-33)。

水中でキャラクターが泳ぐゲームは数多くあります。例えば「スーパーマリオブラザーズ」でも、水中のシーンになると主人公が泳ぎます。地上のジャンプボタンが、水中では泳ぐボタンになります。このようにボタンで泳ぐ操作方法は、多くのゲームで採用されています。

**Fig. 1-31** 水中での左右の動き

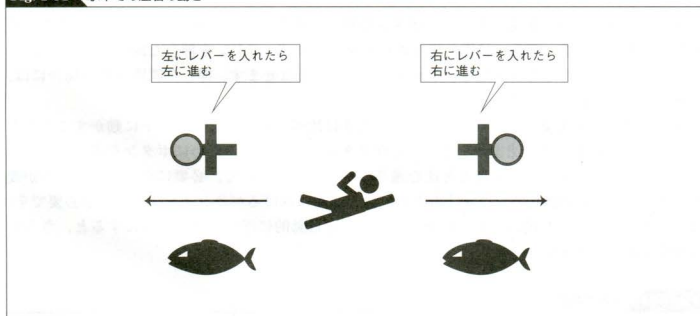



**Fig. 1-32** ボタンを押すと浮かぶ

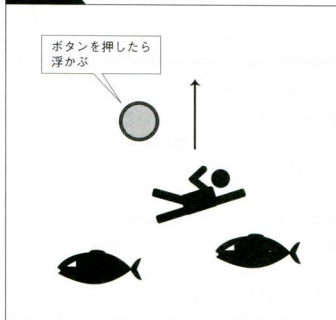
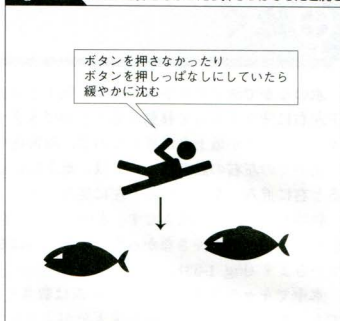
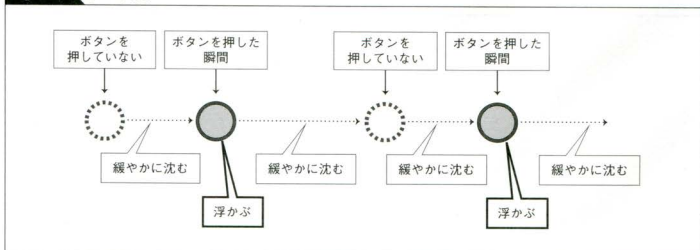



**Fig. 1-33** ボタンを押さなかったり押しっぱなしだと沈む

## アルゴリズム Algorithm

泳ぎのアクションを実現するには、ボタンを押した瞬間に注目します (Fig. 1-34)。ボタンの状態を保存しておき、直前の状態と現在の状態を比べれば、ボタンを押した瞬間がわかります。そして、ボタンを押した瞬間だけ、キャラクターを浮かばせます。押した瞬間以外の場合には、キャラクターを緩やかに沈ませます。

キャラクターを沈ませるときには、浮かぶときに比べて緩やかな速さで下に動かすことがポイントです。あまり急に沈ませると、キャラクターを浮かばせるためにボタンを素早く連打しなくてはなりません。浮かぶ速さと沈む速さのバランスしだいで、必要になる連打の速さが変わります。ゲームの操作性を改善するためには、快適に泳げるバランスを探すことが必要です。また、ボタンを押した後に一定時間キャラクターを自動的に浮かばせるようにすると、さらに操作性がよくなります。

**Fig. 1-34** 泳ぎの実現

## プログラム Program

List 1-7は泳ぎのプログラムです。何も入力しない状態ではキャラクターはゆっくりと下に沈んでいきます。沈む速さと浮かぶ速さを調整してみてください。

**List 1-7** 泳ぐ(CSwimmingManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 左右方向の移動スピード
    float x_speed=0.05f;

    // 上方向の移動スピード
    float up_speed=-0.05f;

    // 下方向の移動スピード
    float down_speed=0.03f;

    // 浮かぶ時間(フレーム数)
    float up_time=20;

    // 左右に関してはレバーを入れた方向に移動する
    VX=0;
    if (is->Left) VX=-x_speed;
    if (is->Right) VX=x_speed;

    // ボタンを押した瞬間ならば、浮かぶ残り時間を設定する
    if (!PrevButton && is->Button[0]) Time=up_time;

    // ボタンを押した瞬間を判定するために、ボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // 上下に関しては、残り時間がある場合には浮かび、
    // 残り時間がない場合には緩やかに沈む
    if (Time>0) {
        VY=up_speed;
        Time--;
    } else {
        VY=down_speed;
    }

    // X座標を更新し、画面からはみ出ないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // Y座標を更新し、画面からはみ出ないように補正する
    Y+=VY;
```

**List 1-7**

```
    if (Y<0) Y=0;
    if (Y>MAX_Y-1) Y=MAX_Y-1;

    // キャラクターの移動方向に応じてグラフィックの向きを変える
    // ReverseXは描画処理から参照する
    if (ReverseX && VX>0) ReverseX=false;
    if (!ReverseX && VX<0) ReverseX=true;

    return true;
}
```

**SAMPLE**

「SWIMMING」は泳ぐアクションのサンプルです。何もしない状態だと、キャラクターはゆっくりと沈んでいきます。ボタンを押すとキャラクターが浮き上がります。左右方向のレバーに合わせて、キャラクターが左右に移動します。沈んでいるときは斜め下方向へ、浮き上がっているときは斜め上方向へ移動します。

**SWIMMING** → p. 392

# ライン移動

奥行き方向に移動するアクションです。2Dグラフィックスで奥行きを表現しているゲームにおいて、手前と奥に移動するために使います。

ライン移動ができるゲームでは、多くの場合、斜めに傾いた絵を斜めにずらして配置することによって、2Dグラフィックスで奥行きを表現しています（Fig. 1-35）。キャラクターが奥行き方向に移動するときには、移動方向を背景の傾きに合わせます（Fig. 1-36）。つまり、奥または手前に移動するときには、画面上では斜めに移動することになります。一方、左右方向に移

**Fig. 1-35** 奥行きの表現

斜めに傾けた絵を
斜めにずらして配置することで
2Dグラフィックスで奥行きを表現する

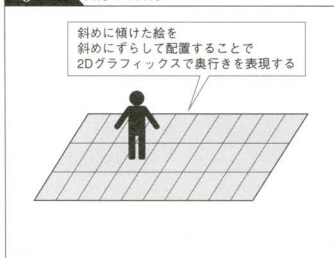
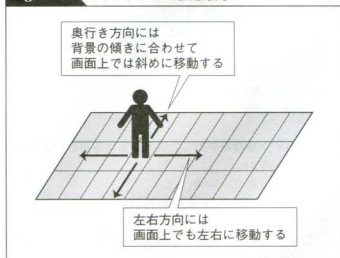



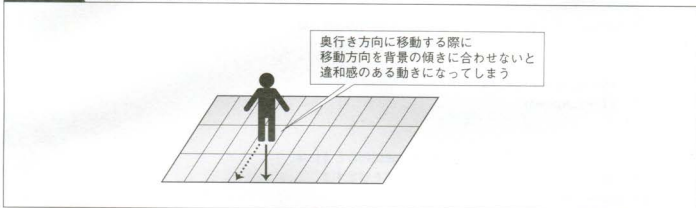
**Fig. 1-36** キャラクターの移動方向

動するときには、普通に画面上でも左右に移動します。

ライン移動は、「ゴールデンアックス」や「ファイナルファイト」といった格闘系のアクションゲームに多く採用されています。画面に奥行きがあり、ライン移動が使えると、敵の後ろに回り込んで攻撃するといった戦術をとることができ、プレイに広がりが出ます。

一方、格闘系以外では「メトロクロス」などもライン移動ができるゲームです。こちらは主に障害物を避けるために、ライン移動を使います。このゲームのように奥行きとライン移動を採用すると、立体感のある魅力的な画面を作ることができます。

## アルゴリズム Algorithm

ライン移動を実現する際のポイントは、奥行き方向の移動時には、背景の傾きに合わせてキャラクターを移動させることです。移動方向を背景の傾きに合わせないと、少し違和感のある動きになってしまいます(Fig. 1-37)。移動方向との絵の傾きがうまく合うように調整しましょう。

**Fig. 1-37** 移動方向を背景の傾きに合わせない場合

## プログラム Program

List 1-8はライン移動のプログラムです。上下のレバーで奥や手前のラインにキャラクターが移動します。移動方向を背景の傾きに合わせるのがポイントです。

**List 1-8** ライン移動(CLineMoveManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 左右方向の移動スピード
    float x_speed=0.07f;

    // 奥行き方向の移動スピード
    float y_speed=0.05f;
```

### List 1-8

```
    // 奥行き方向に移動する際の、
    // 奥行き方向のスピードに対する左右方向のスピードの比率
    // キャラクターを画面上で斜めに移動させるために使う
    float yx_ratio=0.4f;

    // Y座標の最小値
    float min_y=MAX_Y-6*0.8f-0.2f;

    // Y座標の最大値
    float max_y=MAX_Y-1-0.2f;

    // 速度を0にする
    VX=0;
    VY=0;

    // 左右にレバーが入力されていたら、
    // 左右の速度を設定する
    if (is->Left) VX=-x_speed;
    if (is->Right) VX=x_speed;

    // 上にレバーが入力されていて、かつ移動範囲内のときには、
    // 奥へ移動するために斜め右上方向の速度を設定する
    if (is->Up && Y>min_y) {
        VX+=y_speed*yx_ratio;
        VY=-y_speed;
    }

    // 下にレバーが入力されていて、かつ移動範囲内のときには、
    // 手前へ移動するために斜め左下方向の速度を設定する
    if (is->Down && Y<max_y) {
        VX-=y_speed*yx_ratio;
        VY=y_speed;
    }

    // X座標を更新し、画面からはみ出ないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // Y座標を更新する
    // Y座標は速度の設定時に移動範囲内であることを確認しているため、
    // 画面からはみ出ないように補正する必要はない
    Y+=VY;

    return true;
}
```

SAMPLE

「LINE MOVE」はライン移動のサンプルです。上下方向のレバーに合わせて、キャラクターが上下のラインへ移動します。奥行き方向に移動する際には、背景の傾きに合わせて斜め方向へ移動します。

**LINE MOVE** → p. 392

# 画面端ワープ

画面の端を越えて移動すると、画面の反対側の端から出てくるアクションです。画面の端と反対側の端がつながっているような感覚になります。敵が追いかけてくるゲームで画面端ワープができると、画面端に追いつめられても逃げられる可能性が増えて遊びやすくなります。

例えば、画面の左端を越えてキャラクターが移動したときには、画面の右端から出てきます(Fig. 1-38)。また、画面の上端を越えて移動したときには、画面の下端から出てきます(Fig. 1-39)。画面の右端や下端を越えたときにも、それぞれ左端や上端にワープします。

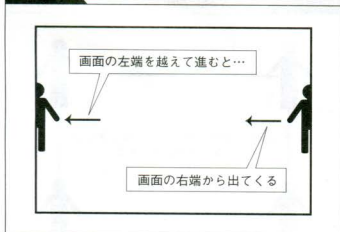
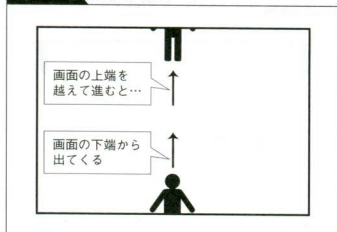
ワープする瞬間には、キャラクターが画面の左右端や上下端に分割されて表示されます。例えば、Fig. 1-39のように画面の上端を越えるときには、画面の上端にはキャラクターの下半身が、下端には上半身が表示されるといった具合です。このようにキャラクターが分割表示されると、キャラクターがいきなり消えてワープするよりも、スムーズな動きになります。

ワープを採用しているゲームには、例えば「パックマン」があります。「パックマン」では画面端にワープトンネルが設置されています。このワープトンネルを上手に使って、モンスターの追跡をかわすわけです。

「パックマン」は明示的にワープトンネルを設置しているゲームですが、特にワープトンネルが表示されていなくても、画面端でワープできるゲームは数多くあります。例えば「アイスクライマー」でも、キャラクターが左右の画面端を越えると、反対側の端から出てきます。

**Fig. 1-38** 画面左右端のワープ

**Fig. 1-39** 画面上下端のワープ

## アルゴリズム　Algorithm

画面端ワープを実現するには、画面の端にキャラクターが到達したときに、画面の反対側から再び出てくるように座標を補正します。それと同時に、画面の両端にキャラクターを表示することによって、ワープ中のキャラクターを表現します。

例えば、画面の左端を越えるときには、画面の左端に加えて、画面の右端にもキャラクターを表示します(Fig. 1-40)。このとき、左右の2つのキャラクターは、画面の幅と同じ長さだけ離して表示します。

同様に画面の上端を越えるときには、画面の上端と下端の両方にキャラクターを表示します(Fig. 1-41)。2つのキャラクターは、画面の高さと同じ長さだけ離して表示します。

少し複雑なのは、画面の左右端と上下端を同時に越えた場合です。このときには、画面の四

**Fig. 1-40** 画面左右端のワープを実現する

画面枠
画面の左端に
キャラクターを表示
画面の右端にも
キャラクターを表示
画面の幅だけ離して
キャラクターを2つ表示する

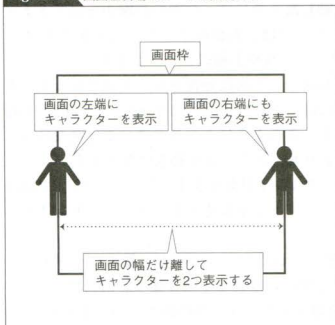



**Fig. 1-41** 画面上下端のワープを実現する

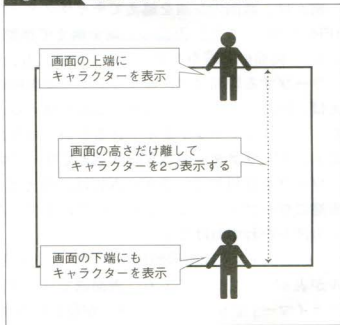
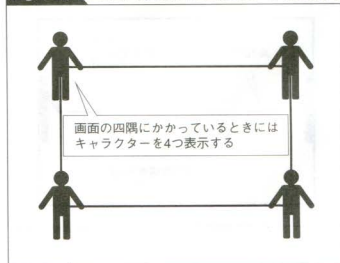



**Fig. 1-42** 画面の四隅にかかったときの見え方

**Fig. 1-43** 画面の四隅にかかったときのワープを実現する

隅にキャラクターが表示された状態になります（Fig. 1-42）。つまり、キャラクターを4つ表示する必要があります（Fig. 1-43）。このとき、左右のキャラクターは画面の幅と同じ長さだけ、上下のキャラクターは画面の高さと同じ長さだけ離して表示します。

このように、キャラクターが画面の端を越えるときには、2つまたは4つのキャラクターを同時に表示する必要があります。実現方法としては、画面の幅と高さの分だけ離して、常に4つのキャラクターを表示しておくのが簡単です（Fig. 1-44）。こうすると、普段は1つのキャラクターしか見えません。ほかの3つのキャラクターは、画面枠の外に隠れています。

**Fig. 1-44** 普段は1つのキャラクターしか見えない

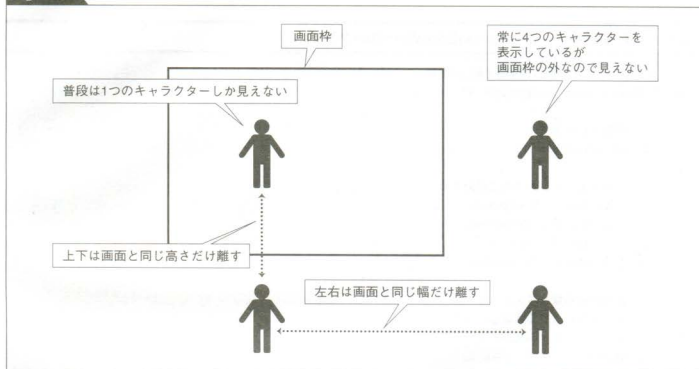
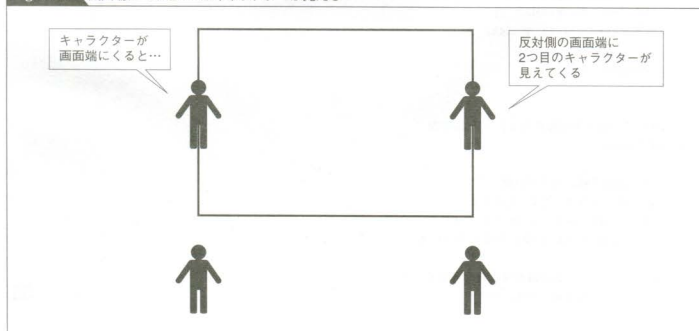



**Fig. 1-45** 画面端にくると2つのキャラクターが見える

キャラクターが画面端にくると、1つのキャラクターが画面内に入ってくるため、合計2つのキャラクターが見えるようになります(Fig. 1-45)。また、キャラクターが画面の四隅にくると、3つのキャラクターが画面内に入ってきて、合計4つのキャラクターが見えるようになります。

## プログラム Program

List 1-9は画面端ワープのプログラムです。キャラクターを移動させる処理と、画面の両端(四隅)にキャラクターを分割して表示する処理がポイントです。

**List 1-9** 画面端ワープ(CScreenEdgeWarpManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.08f;

    // レバーを入力した方向に移動する
    if (is->Left) X-=speed;
    if (is->Right) X+=speed;
    if (is->Up) Y-=speed;
    if (is->Down) Y+=speed;

    // 画面の左右端を越えたときには、
    // 反対側から出てくるように座標を補正する
    if (X<=-1) X+=MAX_X;
    if (X>MAX_X-1) X-=MAX_X;

    // 画面の上下端を越えたときにも、
    // 反対側から出てくるように座標を補正する
    if (Y<=-1) Y+=MAX_Y;
    if (Y>MAX_Y-1) Y-=MAX_Y;

    return true;
}

// キャラクターの描画処理を行うDraw関数
void Draw() {

    // 画面の幅と高さだけ離して、
    // キャラクターを4つ表示する
    for (int i=0; i<2; i++) {
        for (int j=0; j<2; j++) {

            // 元の座標を保存しておく
            float x=X, y=Y;
```

```
            // 座標を変えて、合計4つのキャラクターを描画する
            X+=j*MAX_X;
            Y+=i*MAX_Y;
            CMover::Draw();

            // キャラクターの座標を元に戻す
            X=x;
            Y=y;
        }
    }
}
```

**SAMPLE**

「SCREEN EDGE WARP」は画面端ワープのサンプルです。上下左右のレバーに合わせてキャラクターが移動します。キャラクターが画面端を越えると、反対側の端から現れます。画面端を越えるときは、反対側の端にキャラクターを分割して表示します。四隅を越えるときは、四隅にそれぞれ分割してキャラクターを表示します。

**SCREEN EDGE WARP** → p. 392

## 移動するとライフが減る

移動するたびにライフが減るアクションです。ライフが0になると移動できなくなったり、ミスになって残りキャラクターを1つ失ったりします。ライフがなくならないように、よく考えて行動しなければならないので、バランスしだいではとてもシビアなゲームになります。

ライフが0になると移動できなくなるルールの場合、ライフがある間は自由に移動できます(Fig. 1-46)。ライフがなくなると、移動できなくなります(Fig. 1-47)。

ライフが減るだけではいずれ移動できなくなってしまうので、何かライフを増やす手段も用意されているのが一般的です。例えば、立ち止まっているとライフが回復したり、アイテムを取るとライフが回復したりといったルールが考えられます。また、毒に冒されている一定時間だけ、歩くとライフが減るゲームもあります。

ライフがなくなったときのペナルティも、いろいろなバリエーションが考えられます。移動できなくなったり、ミスになったりする以外に、移動が遅くなったり、魔法が使えなくなったりといったペナルティを課す方法もあるでしょう。

移動するとライフが減るゲームには、「オバケのQ太郎ワンワンパニック」があります。主人公は定期的に食べ物(アイテム)を食べないと、空腹になって動けなくなってしまいます。歩いていてもおなかが減るのですが、飛行するとさらに急激に減っていきます。空腹がある程度に達すると空を飛べなくなるペナルティが課せられ、空腹が極限に達する(ライフが0にな

**Fig. 1-46** ライフがある間は自由に移動できる

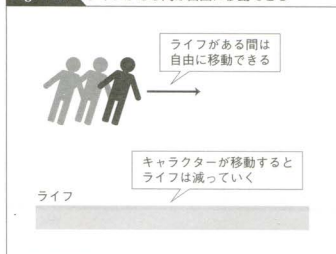
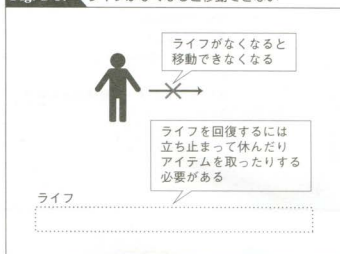



**Fig. 1-47** ライフがなくなると移動できない

る）とミスになってしまいます。

移動するたびにライフが減るゲームは難しくなりがちで、「オバケのQ太郎ワンワンパニック」は特に難易度が高いゲームです。敵や障害物を避けつつ、しかも常に空腹にならないように注意を払わなければならないので、大変歯ごたえのあるゲームになっています。移動するとライフが減るようにする場合には、ゲームバランスを適切に調整する工夫が必要です。

## アルゴリズム Algorithm

移動するとライフが減るようにするには、キャラクターが移動したかどうかをレバー入力や速度などで判定し、移動中ならばライフを減らします。移動中でなければライフを減らしません。立ち止まっているときにライフが回復するゲームでは、ライフを増やします。

ライフを減らす量と増やす量の適切なバランスは、ゲームの内容によって変わります。ライフが速く減るゲームでは、立ち止まったりアイテムを取ったりしたときに、ある程度速くライフを回復させた方がよいでしょう。また、あまりライフの減りが遅いとスリルがなくなってしまいます。移動時にライフを減らすときには、敵や障害物なども総合的に考慮して、バランス調整に時間をかける必要があります。

## プログラム Program

List 1-10は移動するとライフが減るアクションのプログラムです。ライフの減少量と回復量のバランスを調整してみてください。

**List 1-10** 移動するとライフが減る（CDecreaseLifeManクラス）

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {
```

```
    // 移動スピード
    float speed=0.05f;

    // 歩いたときに減るライフの量
    float dec_life=0.005f;

    // 止まっているときに増えるライフの量
    float add_life=0.01f;

    // レバーを入力した方向に進むように速度を設定する
    VX=VY=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;
    if (is->Up) VY=-speed;
    if (is->Down) VY=speed;

    // ライフが0でなければ移動できる
    if (Life>0) {

        // X座標を更新し、画面からはみ出ないように補正する
        X+=VX;
        if (X<0) X=0;
        if (X>MAX_X-1) X=MAX_X-1;

        // Y座標を更新し、画面からはみ出ないように補正する
        Y+=VY;
        if (Y<0) Y=0;
        if (Y>MAX_Y-3) Y=MAX_Y-3;
    }

    // 歩いているときにはライフを減らし、止まっているときには増やす
    if (VX!=0 || VY!=0) Life-=dec_life; else Life+=add_life;

    // ライフが0より小さくなったり、
    // 1より大きくなったりしないように補正する
    if (Life<0) Life=0;
    if (Life>1) Life=1;

    return true;
}
```

SAMPLE

「DECREASING LIFE」は移動によるライフの増減のアクションを行うサンプルです。上下左右のレバーに合わせてキャラクターが移動します。キャラクターの移動中はライフが減少し、停止しているとライフが増加します。ライフがなくなると移動できなくなります。ライフの量は、画面下部のメーターで表します。

**DECREASING LIFE** → p. 392

## 入力と逆の方向へ動く

なんらかの条件で、レバーを入れた方向とは逆の方向にキャラクターが進むようになるアクションです。通常とは逆の向きにキャラクターが動くので、非常に操作が難しくなります。主に、キャラクターに対してなんらかの妨害が行われたときに、こういった状態になります。

例えば、通常は右にレバーを入れるとキャラクターは右に進みます (Fig. 1-48)。しかし、特定のアイテムを取ったり、特定の攻撃を受けたり、特定の罠にかかったりすると、キャラクターの移動方向が逆になります (Fig. 1-49)。移動方向が逆になると、右にレバーを入れているのに、キャラクターは左に進むようになります (Fig. 1-50)。

入力と逆の方向へ動くアクションを採用しているゲームには、例えば「ボンバーマン」シリーズがあります。このゲームではアイテムを取ってキャラクターをパワーアップさせますが、アイテムのなかには取ってはいけないものも含まれています。その1つが、キャラクターの移動方向を逆にしてしまうアイテムです。アイテムを素早く取らなければならないというゲームの性質を逆手に取って、こういった罠のアイテムが交ぜてあるというわけです。

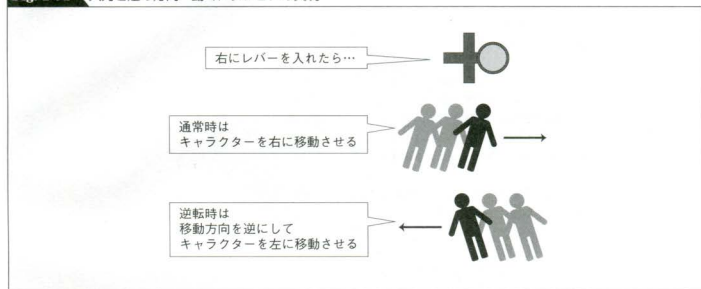
**Fig. 1-48** 通常の動き

**Fig. 1-49** 移動方向が逆になるアイテム

**Fig. 1-50** 移動方向が逆になったときの動き

## アルゴリズム Algorithm

入力と逆の方向へ動くアクションを実現するには、キャラクターの移動方向が通常か逆転かを判別するフラグを用意しておきます。そして、通常時と逆転時でキャラクターの移動方向を逆にします。例えば、レバーが右に入力されたときに、通常時はキャラクターを右に移動させ、逆転時は左に移動させます（Fig. 1-51）。

**Fig. 1-51** 入力と逆の方向へ動くアクションの実現

## プログラム Program

List 1-11は入力と逆の方向へ動くアクションのプログラムです。画面上のアイテムに触れることで、レバーの操作とキャラクターの移動方向が反転します。

**List 1-11** 入力と逆の方向へ動く（CReverseDirectionManクラス）

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.1f;

    // 移動方向が逆になっていたら、移動スピードの符号を反転する
    // Reverseはキャラクターの移動方向が通常か逆転かを表すフラグ
    if (Reverse) speed=-speed;

    // レバーの入力に応じてキャラクターを移動させる
    if (is->Left) X-=speed;
    if (is->Right) X+=speed;
```

**List 1-11**

```
	if (is->Up) Y-=speed;
	if (is->Down) Y+=speed;

	// 画面の端から出ないようにX座標を補正する
	if (X<0) X=0;
	if (X>MAX_X-1) X=MAX_X-1;

	// 画面の端から出ないようにY座標を補正する
	if (Y<0) Y=0;
	if (Y>MAX_Y-1) Y=MAX_Y-1;

	// アイテムを取ったかどうかのフラグ
	bool get_item=false;

	// アイテムを取ったかどうかの判定
	// すべてのアイテムについて調べる
	// アイテムを取るたびに、移動方向の通常と逆転が入れ替わる
	for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
		CMover* mover=(CMover*)i.Next();

		// アイテムとキャラクターが一定距離よりも近づいたら、
		// アイテムを取ったと判定する
		if (
			mover->Type==1 &&
			abs(mover->X-X)<1.0f &&
			abs(mover->Y-Y)<1.0f
		) {
			// 移動方向を反転させる
			// 移動方向が正しいときには逆に、逆のときには正しくする
			Reverse=!Reverse;

			// アイテムを消去する
			i.Remove();

			// アイテムを取ったフラグを立てる
			get_item=true;
		}
	}

	// アイテムを取ったら、新しいアイテムを1つ出現させる
	if (get_item) new CReverseDirectionItem();

	// 逆方向に進んでいるときには、
	// キャラクターの上下を逆向きに表示する
	Angle=Reverse?0.5f:0;

	return true;
}
```

SAMPLE

「REVERSED DIRECTION」はアイテムによって移動方向が逆転するアクションのサンプルです。画面内にはキャラクターと複数のアイテムが表示されています。キャラクターがアイテムに接触すると移動方向が逆転します。再びアイテムに接触すると、元に戻ります。逆転中は、わかりやすいようにキャラクターを上下反転して表示しています。また、キャラクターと接触したアイテムは消去し、別の位置に新しいアイテムを1つ出現させています。

**REVERSED DIRECTION** → p. 393

## ループ

円形のループをキャラクターが回転しながら走り抜けるアクションです。宙返りジェットコースターのようなダイナミックな動きが楽しめます。

ループのアクションは、キャラクターがループに近づくところから始まります(Fig. 1-52)。キャラクターがループの真下付近にくると、ループに進入します(Fig. 1-53)。

ループに入ると、キャラクターはループを回り始めます(Fig. 1-54)。キャラクターはどんどんループを回っていき(Fig. 1-55)、ループを一回転するまで回り続けます。

ループを1回転したら、ループのアクションは終わりです(Fig. 1-56)。キャラクターはループを抜けて、通常の移動方法に戻ります(Fig. 1-57)。

ループを採用しているゲームの例としては、「ソニック・ザ・ヘッジホッグ」シリーズがあります。このゲームはスピーディーなキャラクターの動きが見どころですが、ループを使うことによって、アクションをより一層ダイナミックにしています。高速でループを駆け抜けていくキャラクターの動きは、見ているだけでも爽快感があります。

**Fig. 1-52** ループに近づく

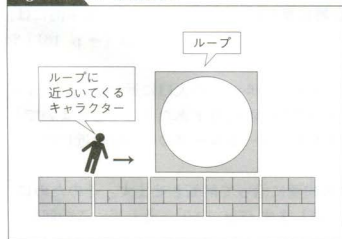



**Fig. 1-53** ループに入る

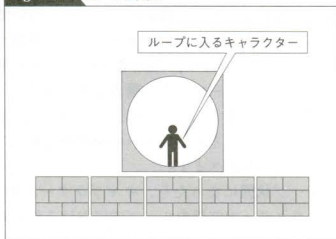



Fig. 1-54 ループを回り始める

Fig. 1-55 さらにループを回る

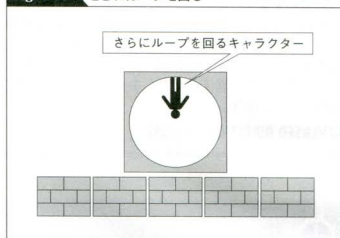



Fig. 1-56 ループを終える

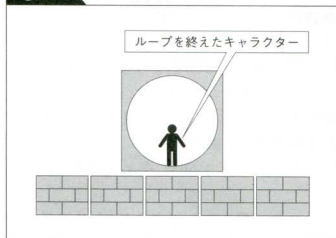
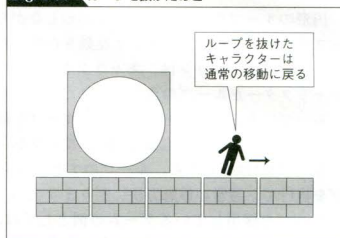



Fig. 1-57 ループを抜けたあと

## アルゴリズム Algorithm

ループを実現するには、通常状態とループ状態を区別して、それぞれ異なる方法でキャラクターを移動させます。通常状態の移動については、特に難しいことはありません。基本的には、レバーを入力した方向にキャラクターを移動させるだけです。「レバーダッシュ（→ p. 16）」や「ボタンダッシュ（→ p. 20）」などの項目を参照してください。

ポイントはループ状態の動きです。まず、キャラクターがループの入口に近づいたら、ループ状態に入ります（Fig. 1-58）。ループの入口は、ループの中心の真下あたりにするとよいでしょう。ループとキャラクターの座標を比較して、キャラクターがループの中心線付近に近づいたかどうかを調べます。

ループ状態になったら、キャラクターをループの内壁に沿って回転させます。そのために、キャラクターの位置をループ内における回転角度で表します（Fig. 1-59）。そして、この回転角度を変化させることによって、キャラクターにループを回らせます。

キャラクターの座標は、ループの中心座標・回転半径・回転角度から計算できます。例えば、中心座標を（lx, ly）、回転半径をradius、回転角度をangleとすると、キャラクターの座標

(x, y) は、

```
x=lx+radius＊cos((-angle+0.25)＊π＊2)
y=ly+radius＊sin((-angle+0.25)＊π＊2)
```

となります。πは円周率を表します。なお、ここではangleの範囲を0（0度）から1（360度）とし、回転開始時のangleの値を0としています。

ループ状態では、ループの角度に合わせてキャラクターも回転表示する必要があります。これは、キャラクターがループの内壁に沿って駆け抜けていく様子を表現するためです。

キャラクターがループを回り切ったら、ループ状態を終えて通常状態に戻ります。これは回転角度が360度に達したかどうかで判定することができます（Fig. 1-60）。前述のようにangleの範囲を0から1にしたときには、angleが1に達したときがループを回り終えたときです。なお、回転角度の上限を360度ではなく720度や1080度にすると、2周または3周しないと抜けられないループを作ることができます。

**Fig. 1-58** ループ状態の開始

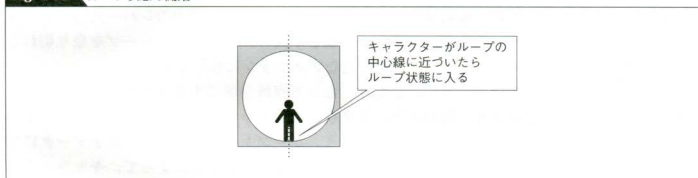



**Fig. 1-59** ループ中の座標計算

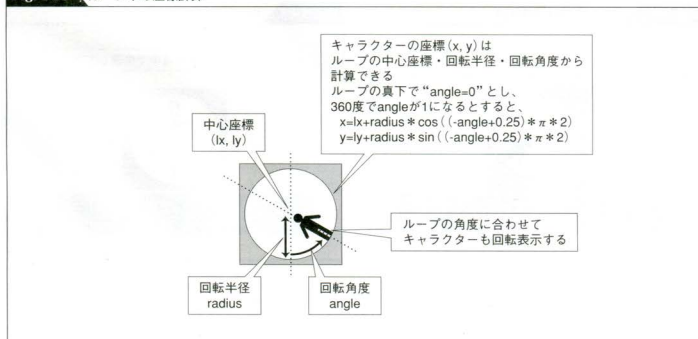



**Fig. 1-60** ループ状態の終了

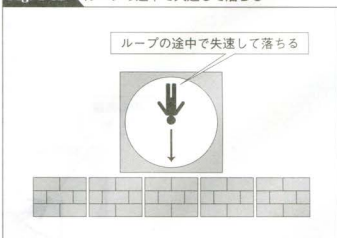
## ループの応用

ループの応用としては、ループへの進入速度が足りなかったときに、ループを登り切れずに戻ってくるアクションなどがあります（Fig. 1-61）。また、ループの途中で失速してしまったときに、遠心力が足りなくてループから落下するアクションなども面白いでしょう（Fig. 1-62）。「ソニック・ザ・ヘッジホッグ」にはこれらのアクションも導入されています。

これらのアクションを実現するには、ループへの進入速度やループ中のレバー入力から、ループ中の速度を計算します。そして、速度が一定値を下回ったときには、ループを登り切れなくて戻ったり、ループから落ちたりといった特別なアクションに移行します。

また、ループの形を円ではなく楕円にしたり、もっと複雑な形にすることもできます。ループを複雑な形にすることにより、円のように単純な式ではキャラクターの位置が計算できない場合には、位置をデータにしておくとよいでしょう。ループ上のいくつかの座標をデータにしておき、それらの座標をスプライン曲線などで滑らかに接続することによって、キャラクターの座標を計算する方法もあります。

**Fig. 1-61** ループを登り切れない

**Fig. 1-62** ループの途中で失速して落ちる

##  プログラム Program

List 1-12はループのプログラムです。ループ中のキャラクターの座標と向きがポイントです。また、ループの大きさやループ中の速度などを調整してみてください。

**List 1-12** ループ(CLoopManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 通常時の移動スピード
    float speed=0.2f;

    // ループ状態の移動スピード(角度の変化)
    float loop_speed=0.01f;

    // ループ中心点のY方向のオフセット
    float loop_y=0.5f;

    // ループの半径
    float loop_radius=2.4f;

    // ループ状態の処理
    if (Loop) {

        // LoopAngleはループ内におけるキャラクターの位置(角度)を表す
        // ここでは現在の位置からキャラクターの画面上の座標を計算する
        X=Loop->X+loop_radius*cosf((-LoopAngle+0.25f)*D3DX_PI*2);
        Y=Loop->Y+loop_y+loop_radius*sinf((-LoopAngle+0.25f)*D3DX_PI*2);

        // キャラクターの位置(角度)を更新する
        LoopAngle+=loop_speed;

        // ループを回り終えたときの処理
        // キャラクターをループの真下の地面に配置し、通常状態へ移行する
        if (LoopAngle>1) {
            X=Loop->X+speed;
            Y=MAX_Y-2;
            Loop=NULL;
        }

        // ループ内の位置に合わせて、キャラクターを回転させて表示する
        Angle=-LoopAngle;
    } else

    // 通常状態の処理
    {
        // レバーを右に入れたら右に進む
```

**List 1-12**

```
		VX=0;
		if (is->Right) VX=speed;

		// ループに入ったかどうかを判定する
		// すべてのループについて当たり判定処理を行う
		for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
			CMover* mover=(CMover*)i.Next();

			// ループの真下に一定距離まで近づいたら、
			// ループに入ったとして、ループ状態に移行する
			if (
				mover->Type==1 &&
				X<mover->X &&
				X>=mover->X-speed
			) {
				VX=0;
				Loop=mover;
				LoopAngle=0;
				break;
			}
		}

		// 通常状態では、速度に応じてキャラクターを傾けて表示する
		Angle=VX/speed*0.05f;
	}

	return true;
}
```

**SAMPLE**

「LOOP」はループのアクションのサンプルです。右方向にレバーを入れるとキャラクターが前進します。移動方向にループ台が用意されていて、進入するとキャラクターがループに沿って回転します。ループを1周すると、キャラクターは再び床の上を直線的に移動するようになります。

**LOOP** → p. 393

## まとめ Stage01

本章ではアクションゲームにおいて最も基本的な「移動」について解説しました。レバーを入れた方向にキャラクターが移動する、というのが基本ですが、細かく見ていくとかなりいろいろな方式があります。どの方式も、キャラクターの動きをより面白く、より魅力的にするために考え出されたものです。

というわけで、「キャラクターが滑らかに、気持ちよく移動することがアクションゲームの基本！」というのが本章のまとめです。

「ジャンプ」は、アクションゲームにおいて移動に次いで基本的なアクションです。主人公が人や動物で、横方向から見る視点のゲームでは、ほとんどの場合キャラクターをジャンプさせることができます。このジャンプにも、実はいろいろなバリエーションがあります。

# ジャンプ

## Jump

ActionGame Algorithm Maniax

# 固定長ジャンプ

ボタンを押すと、キャラクターがジャンプするアクションです。ゲームによっては、レバーを上や斜め上に入れたときにジャンプするものもあります。多くの場合、キャラクターは滑らかな放物線を描いてジャンプします。

ジャンプを採用したゲームには、ジャンプ中にボタンを放すことによってジャンプの高さを調整できるものと、ジャンプの高さが固定されているものとがあります。本書では前者を「可変長ジャンプ」、後者を「固定長ジャンプ」として、別々に解説することにしました。ここではまず、固定長ジャンプについて説明します。

簡単のために、最初は静止しているキャラクターがジャンプする場合を考えてみましょう(Fig. 2-1)。ボタンを押すと、キャラクターはジャンプを開始し、垂直に上昇を始めます(Fig.

Fig. 2-1 静止しているキャラクター

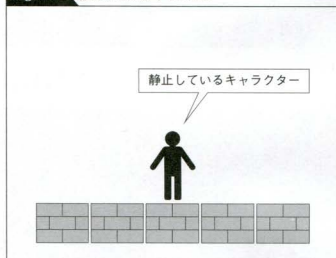



Fig. 2-2 ジャンプの開始

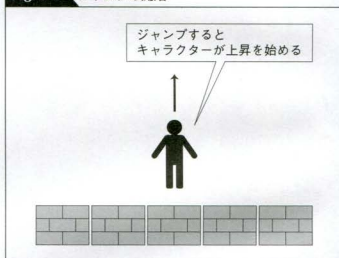



Fig. 2-3 上昇スピードが落ちる

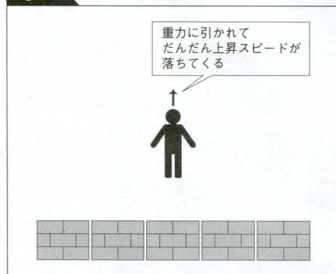
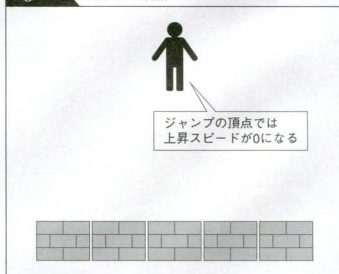



Fig. 2-4 ジャンプの頂点

2-2)。

キャラクターは上昇していきますが、重力に引かれて、だんだん上昇スピードが落ちてきます（Fig. 2-3)。そしていつかは、上昇スピードが0になります（Fig. 2-4)。ここがジャンプの頂点です。

頂点に達したあとは、上昇から落下に移行します（Fig. 2-5)。落下中は、重力に引かれてだんだん落下スピードが大きくなります。そして、地面に着地したらジャンプは終了です（Fig. 2-6)。

次に、左右に移動しながらジャンプした場合について考えます。縦方向の動きは、静止しているキャラクターがジャンプした場合と同じです。これに横方向の動きが加わることによって、ジャンプの軌跡は放物線になります（Fig. 2-7)。

ジャンプを使ったゲームは非常に数多くあります。例えば「ドンキーコング」は、転がってくるタルをジャンプで避けながら進んでいくゲームです。このゲームのジャンプは固定長ジャ

**Fig. 2-5** 落下する

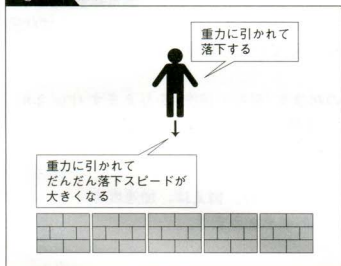



**Fig. 2-6** ジャンプの終了

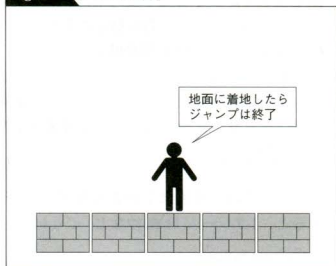
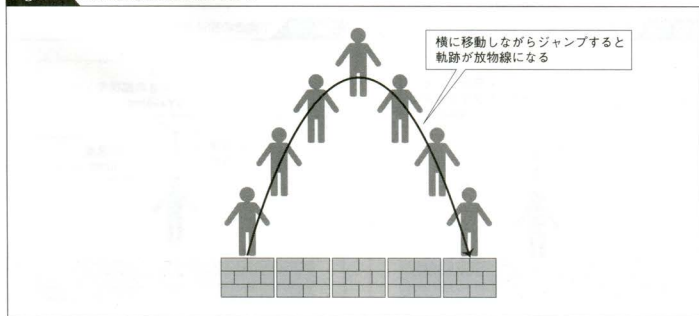



**Fig. 2-7** 左右に移動しながらのジャンプ

ンプなので、ボタンを放すことによってジャンプの高さを調整することはできません。ジャンプを開始したら、ジャンプが終了するまでは操作できなくなります。

固定長ジャンプよりも可変長ジャンプの方が、プレイヤーが自由にジャンプできて快適なように思えますが、必ずしもそうではありません。「ドンキーコング」の場合には、タルをぎりぎり跳び越えられるような、ちょうどよいジャンプの高さにあらかじめ調整がされています。そのため、プレイヤーはジャンプ中にボタンを放すことについては考える必要がなく、ジャンプをする瞬間のタイミングだけを見計らってボタンを押すだけですみます。結果として、リズミカルにジャンプボタンを押して次々にタルを跳び越えていくような、軽快なゲーム性が実現されています。

## アルゴリズム Algorithm

固定長ジャンプを実現するには、ジャンプ状態と通常状態を区別します。通常状態では、キャラクターはレバーで左右に動くことができます。ジャンプボタンを押したら、ジャンプ状態に入ります。ジャンプの開始時にキャラクターが静止していたら垂直に跳ぶジャンプになり、左右に動いていたら放物線のジャンプになります。

ジャンプを開始したら、キャラクターに上向きの初速度（最初の速度）を与えます（Fig. 2-8）。例えば、キャラクターの速度をVYとすると、VYに一定値jump_speedを設定します。

```
VY=jump_speed
```

ジャンプ中は、速度に下向きの加速度を加えます（Fig. 2-9）。例えば、加速度をjump_accelとすると、1フレーム（1/60秒など）ごとにVYにjump_accelを加えます。

```
VY+=jump_accel
```

下向きの加速度が加わることによって、ある時点でキャラクターは上昇から落下に移行しま

Fig. 2-8 初速度を与える

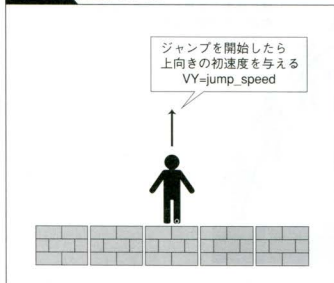
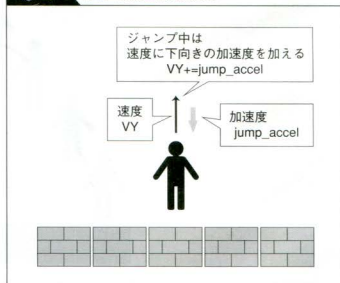



Fig. 2-9 下向きの加速度を加える

す。ここがジャンプの頂点です (Fig. 2-10)。頂点の高さは、初速度 (jump_speed) と加速度 (jump_accel) のバランスによって決まります。初速度が大きいほどジャンプは高く、小さいほど低くなります。また、加速度が大きいほど素早いジャンプになり、小さいほどゆったりしたジャンプになります。例えば、初速度が大きくて加速度が小さいと、緩やかで高いジャンプになります。

キャラクターが着地したら、ジャンプは終了です。ジャンプ状態から通常状態に戻ります。ジャンプ状態では、レバーで左右に動いたり、ジャンプをしたりといったことはできません。通常状態に戻ることで、再び左右に動いたり、ジャンプをしたりすることができるようになります。

地面に着地したかどうかを判定するには、キャラクターのY座標を調べるか、地面との当たり判定処理を行います (Fig. 2-11)。地面の高さが決まっているならば、Y座標を調べるのが簡単でしょう。いろいろな高さの地面があるならば、地面との当たり判定処理を行う必要があります。

左右に移動しながらジャンプした場合も、Y方向の速度については垂直ジャンプと同じ計算方法です (Fig. 2-12)。X方向 (横方向) については、ジャンプ開始時の速度を保ちます。例えば、ジャンプ開始時にX方向の速度がVXならば、ジャンプ中も速度はVXのままで左右に移動します。

**Fig. 2-10** 上昇から落下に移行する

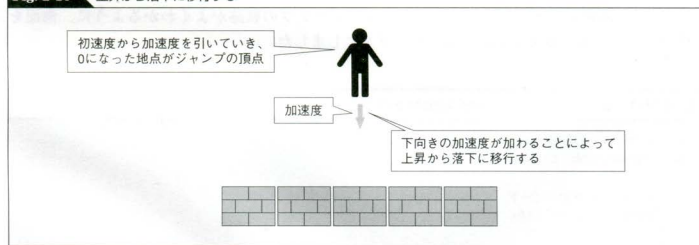



**Fig. 2-11** ジャンプの終了を判定する

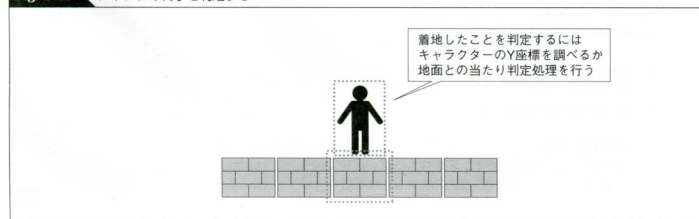
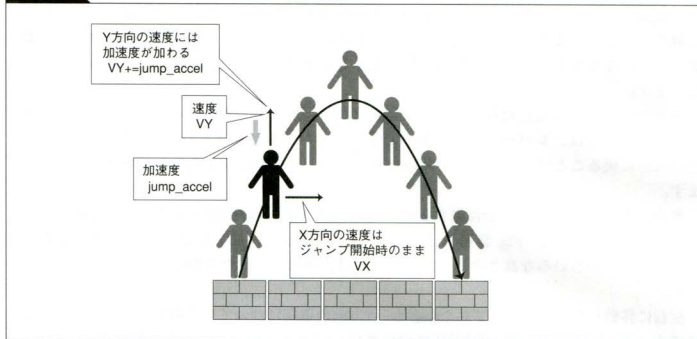



**Fig. 2-12** 左右に移動しながらジャンプした場合の速度

## プログラム Program

List 2-1は固定長ジャンプのプログラムです。ジャンプの軌跡がよくわかるように、画面を左右にスクロールさせない固定画面のサンプルにしました。

**List 2-1** 固定長ジャンプ(CFixedJumpManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.15f;

    // ジャンプの初速度
    float jump_speed=-0.5f;

    // ジャンプの加速度
    float jump_accel=0.02f;

    // 地面にキャラクターがいるときのY座標
    // 着地の判定に使う
    float ground_y=MAX_Y-2;

    // 通常状態の処理
    // Jumpはジャンプ状態を表すフラグ
    // trueのときはジャンプ状態、falseのときは通常状態を示す
    if (!Jump) {
```

```
        // レバーの入力にしたがってX方向の速度を設定する
        // VXはキャラクターのX方向の速度
        VX=0;
        if (is->Left) VX=-speed;
        if (is->Right) VX=speed;

        // ボタンを押したらジャンプ状態に移行する
        // ジャンプ状態のフラグと初速度を設定する
        // VYはキャラクターのY方向の速度
        if (is->Button[0]) {
            Jump=true;
            VY=jump_speed;
        }
    } else

    // ジャンプ状態の処理
    {
        // Y方向の速度に加速度を加える
        VY+=jump_accel;

        // Y座標の更新
        Y+=VY;

        // 着地の判定
        // キャラクターが落下中(Y方向の速度が正の値)で、
        // かつ地面にキャラクターがいるときのY座標に達していたら、
        // 着地したと判定する
        // ジャンプ状態のフラグをfalseにして通常状態に戻り、
        // Y座標を地面にいるときの座標に設定する
        if (VY>0 && Y>=ground_y) {
            Jump=false;
            Y=ground_y;
        }
    }

    // X座標を更新し、画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「FIXED JUMP」は固定長ジャンプのサンプルです。左右のレバーでキャラクターが移動し、ボタンでジャンプします。キャラクターが静止している状態でボタンを押せば垂直に、左右に移動中にボタンを押せば放物線状にジャンプします。ジャンプの高さは固定されています。

**FIXED JUMP** → p. 393

# 可変長ジャンプ

ボタンを押すと、キャラクターがジャンプするアクションです。ジャンプの上昇中にボタンを放すと、キャラクターは上昇を中断して落下します。ボタンを押す長さを調整することによって、ジャンプの高さを制御することができます。

最初は静止しているキャラクターがジャンプする場合を考えてみましょう(Fig. 2-13)。ボタンを押すとキャラクターはジャンプを開始し、垂直に上昇を始めます(Fig. 2-14)。これは「固定長ジャンプ(→ p. 60)」の場合と同じです。

ボタンを押しているかぎりは、固定長ジャンプと同じ動きになります。キャラクターの上昇スピードは、重力に引かれてだんだん落ちてきます(Fig. 2-15)。

上昇中にボタンを放すと、上昇を中断して落下に移行します(Fig. 2-16)。ボタンを放した瞬間の位置がジャンプの頂点になるので、ボタンを押しっぱなしにしたときよりも低いジャンプになります。ボタンをずっと押しっぱなしにしたときには、固定長ジャンプと同じ高さのジャンプになります。

落下に移行したあとの動きは、固定長ジャンプの場合と同じです。重力に引かれて、だんだん落下スピードが大きくなります(Fig. 2-17)。地面に着地したらジャンプは終了です(Fig. 2-18)。

**Fig. 2-13** 静止しているキャラクター

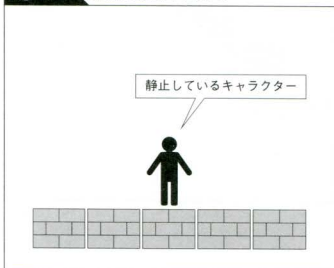



**Fig. 2-14** ジャンプの開始

ボタンを放したときのジャンプの軌跡と、放さなかったときの軌跡を比べてみましょう（Fig. 2-19）。上昇時にボタンを放したときには、放さなかったときよりも低いジャンプになります。ボタンをタイミングよく放すことで、ジャンプの高さを自由に調整することが可能です。

可変長ジャンプを使ったゲームも非常に数多くあります。例えば「スーパーマリオブラザーズ」にも、可変長ジャンプが採用されています。このゲームでは左右方向の速度も可変なので、キャラクターの速度とボタンのタイミングによって、ジャンプの距離が複雑に変化します。この複雑さがゲームの難易度を高めているところであり、面白いところでもあります。

**Fig. 2-15** ボタンを押しているときの動き

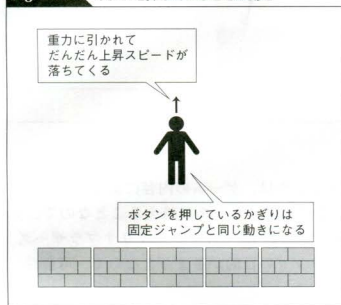
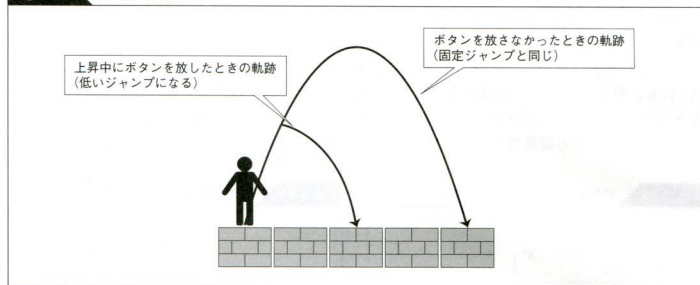



**Fig. 2-16** ボタンを放したときの動き

**Fig. 2-17** 落下スピードが大きくなる

**Fig. 2-18** ジャンプの終了

**Fig. 2-19** ジャンプの軌跡の比較

## 固定長と可変長

固定長ジャンプと可変長ジャンプのどちらが適切なのかは、ゲームの内容によって変わります。「ドンキーコング」の場合には、ジャンプの主な目的がタルを跳び越えることなので、固定長ジャンプの方がシンプルで軽快なゲームになるでしょう。「スーパーマリオブラザーズ」の場合には、崖を跳び越えたり敵を踏んだりと、ジャンプの目的がさまざまなので、可変長ジャンプの方が快適に遊ぶことができます。

固定長ジャンプに比べて、可変長ジャンプは自由度が高いぶん、プレイヤーはジャンプ中のキャラクターのコントロールに注意を払う必要があります。逆に固定長ジャンプは、ジャンプをする瞬間には注意を払いますが、一度ジャンプしてしまったらコントロールが利かないので、ジャンプを終えたあとのことを考えることができます。

また、固定長ジャンプには軌跡が読みやすいという利点もあります。この利点は、例えば格闘ゲームで生かすことができます。格闘ゲームでは、ジャンプを固定長ジャンプにしておくと、相手のジャンプの軌跡が読めるため、反撃や防御といった対応がスムーズにできて面白くなります。

## アルゴリズム Algorithm

可変長ジャンプを実現するには、ジャンプの上昇中にボタンの状態を調べます。上昇中にボタンを放したら、Y方向の速度を反転させるか、あるいは0にします(Fig. 2-20)。Y方向の速度をVYとすると、速度を反転させるには、

```
VY=-VY
```

とします。速度を0にするには、

```
VY=0
```

とします。

速度を反転させた場合、ジャンプの軌跡はFig. 2-21のようになります。頂点をはさんで、ジャンプの軌跡が対称になることが特徴です。

速度を0にした場合は、ジャンプの軌跡がFig. 2-22のようになります。上昇時に比べて、落下時の移動距離が長くなりますが、ジャンプの動きが滑らかになることが特徴です。

どちらの方法を使っても、可変長ジャンプを実現することができます。両者の動きの違いは微妙なので、実際に動きを見比べてみて、ゲームの内容にふさわしい方を採用するとよいでしょう。

Fig. 2-20　可変長ジャンプの実現方法

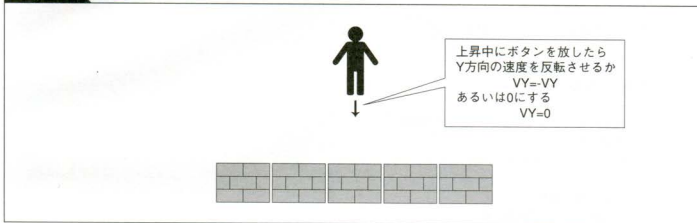



Fig. 2-21　速度の方向を反転させた場合の軌跡

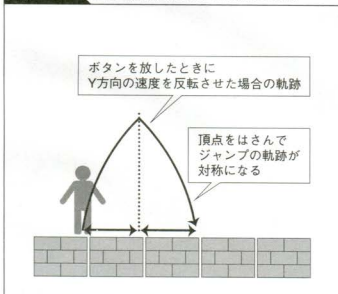
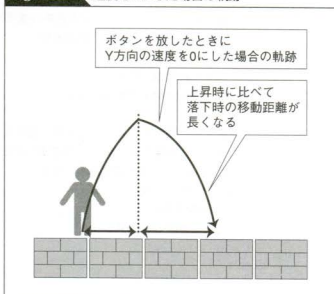



Fig. 2-22　速度を0にした場合の軌跡

## プログラム Program

List 2-2は可変長ジャンプのプログラムです。固定長ジャンプとの違いは、ジャンプ中にボタンを放したときの処理だけです。List 2-1と見比べてみてください(→ p. 64)。

**List 2-2** 可変長ジャンプ（CVariableJumpManクラス）

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.15f;

    // ジャンプの初速度
    float jump_speed=-0.5f;

    // ジャンプの加速度
    float jump_accel=0.02f;

    // 地面にキャラクターがいるときのY座標
    // 着地の判定に使う
    float ground_y=MAX_Y-2;

    // 通常状態の処理
    // Jumpはジャンプ状態を表すフラグ
    // trueのときはジャンプ状態、falseのときは通常状態を示す
    if (!Jump) {

        // レバーの入力にしたがってX方向の速度を設定する
        // VXはキャラクターのX方向の速度
        VX=0;
        if (is->Left) VX=-speed;
        if (is->Right) VX=speed;

        // ボタンを押したらジャンプ状態に移行する
        // ジャンプ状態のフラグと、初速度を設定する
        // VYはキャラクターのY方向の速度
        if (is->Button[0]) {
            Jump=true;
            VY=jump_speed;
        }
    } else

    // ジャンプ状態の処理
    {
        // Y方向の速度に加速度を加える
        VY+=jump_accel;
```

```
        // Y座標の更新
        Y+=VY;

        // 着地の判定
        // キャラクターが落下中(Y方向の速度が正の値)で、
        // かつ地面にキャラクターがいるときのY座標に達していたら、
        // 着地したと判定する
        // ジャンプ状態のフラグをfalseにして通常状態に戻り、
        // Y座標を地面にいるときの座標に設定する
        if (VY>0 && Y>=ground_y) {
            Jump=false;
            Y=ground_y;
        }

        // 上昇時にボタンを放したときの処理
        // この処理だけが固定長ジャンプの場合と異なる
        // キャラクターが上昇中(Y方向の速度が負の値)で、
        // かつボタンを放していたら、
        // Y方向の速度を0にすることによって、落下に移行する
        // Y方向の速度を反転させてもよい(VY=0のかわりにVY=-VYとする)
        if (VY<0 && !is->Button[0]) {
            VY=0;
        }
    }

    // X座標を更新し、画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

**SAMPLE**

「VARIABLE JUMP」は可変長ジャンプのサンプルです。左右のレバーでキャラクターが移動し、ボタンでジャンプします。キャラクターが静止している状態でボタンを押せば垂直に、左右に移動中にボタンを押せば放物線状にジャンプします。ボタンを放すタイミングでジャンプの高さを調整することができます。

**VARIABLE JUMP** → p. 393

# 2段ジャンプ

ジャンプの頂点付近でボタンを押すと、キャラクターが空中で再びジャンプするアクションです。2段ジャンプでは、最初のジャンプの頂点でもう一度ジャンプすることができます。3段ジャンプや4段ジャンプも実現できますが、多くのゲームは2段ジャンプまでを採用しています。

左右に進んでいるキャラクターが1段ジャンプする場合を考えてみましょう（Fig. 2-23）。地上でボタンを押すと、キャラクターがジャンプします。キャラクターは放物線を描き、着地するとジャンプが終わります。

2段ジャンプの場合には、地上でボタンを押してジャンプしたあとに、ジャンプの頂点付近でもう一度ボタンを押します（Fig. 2-24）。すると、キャラクターが空中で再びジャンプして、1段ジャンプのときよりも高く跳躍することができます。

2段ジャンプを採用したゲームには、「ドラゴンバスター」などがあります。「ドラゴンバスター」の場合は、レバーを上に入れるとキャラクターがジャンプします。ジャンプの頂点付近でもう一度レバーを上に入れると、キャラクターが2段ジャンプします。2段ジャンプすることによって、敵を跳び越えたり、高いところにいる敵を攻撃したりすることができます。

2段ジャンプと可変長ジャンプは、ジャンプの高さを調整するという点では似ています。しかし、2段ジャンプはタイミングよく2段目のジャンプボタンを入力する必要があるため、可変長ジャンプに比べると操作の難易度は上がります。

**Fig. 2-23** 1段ジャンプの軌跡

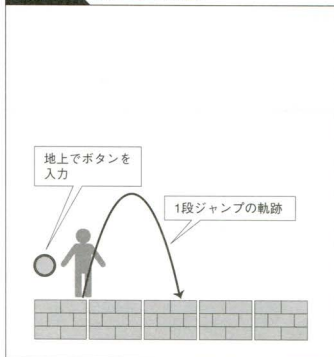
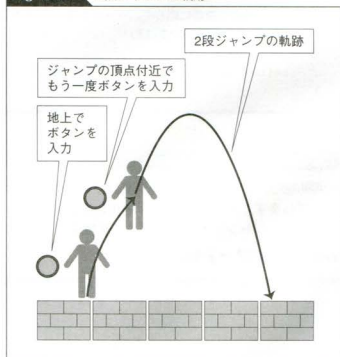



**Fig. 2-24** 2段ジャンプの軌跡

## アルゴリズム Algorithm

2段ジャンプを実現する際のポイントは、2段ジャンプのボタン入力を受け付けるタイミングです。一般に2段ジャンプでは、2段目のボタンをいつ入力してもよいわけではありません。ジャンプの頂点付近で入力したときだけ、2段ジャンプになります。

そのため、キャラクターがジャンプの頂点付近にいるかどうかを判定して、2段ジャンプの入力を受け付ける必要があります（Fig. 2-25）。入力を受け付けるのは、ジャンプの頂点付近の特定の範囲だけです。

ジャンプの頂点付近かどうかを判定するために、ここではキャラクターのY方向の速度に注目することにしました（Fig. 2-26）。速度の大きさはジャンプした瞬間が一番大きく、ジャンプの頂点に近づくにしたがって、重力に引かれてだんだん小さくなります。そして、ジャンプの頂点で速度の方向が反転して、今度は重力に引かれてだんだん速度が大きくなります。

**Fig. 2-25** 2段ジャンプのボタン受け付け範囲

特定の範囲でボタンを入力すると
2段ジャンプになる

2段ジャンプのボタン入力を
受け付ける範囲

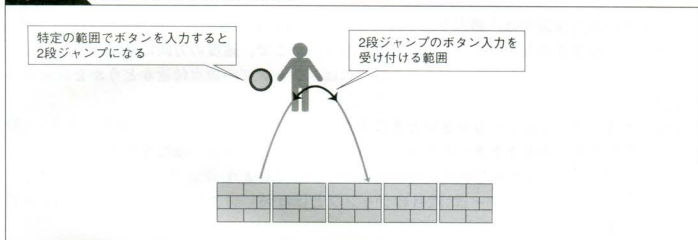



**Fig. 2-26** ボタン受け付け範囲と速度の関係

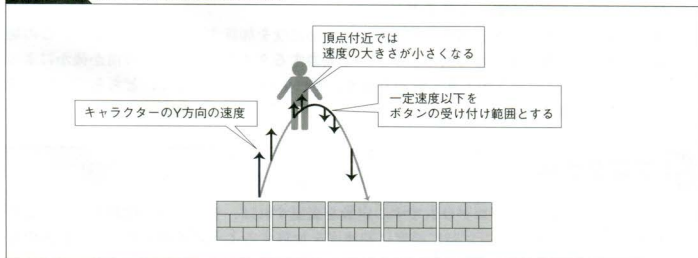
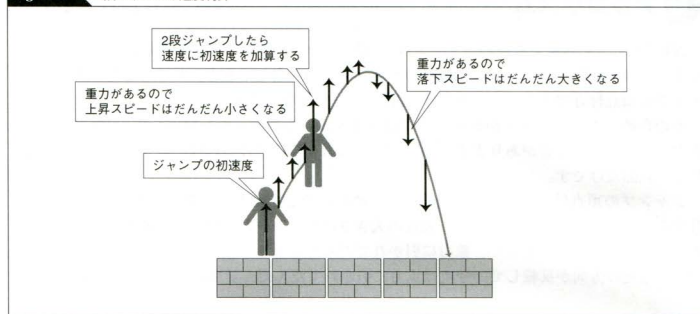



**Fig. 2-27** 2段ジャンプの速度制御

ジャンプの頂点付近では、頂点をはさんで速度の方向が上向きから下向きに変わりますが、いずれにしても速度の大きさは小さくなっています。そこで、速度の方向にかかわらず、速度の大きさが一定値よりも小さいかどうかを調べれば、ジャンプの頂点付近かどうかを判定することができます。

速度の大きさが一定値よりも小さいときにボタンを押したら、2段ジャンプを行います。2段ジャンプをしたら、キャラクターをジャンプさせるために、上向きの速度を与える必要があります。例えば、ジャンプの初速度を再度設定するとよいでしょう（Fig. 2-27）。初速度を設定することによって、キャラクターは1段目のジャンプとほぼ同じ高さだけ、さらに跳び上がってから落下します。

この方式では、ジャンプの頂点ぴったりでボタンを入力した場合に、最も高く跳ぶことができます。頂点の少し前や、頂点の少しあとでボタンを押したときには、頂点との高さの差分だけ、ジャンプ全体の高さが低くなります。

初速度を設定するかわりに、現在の速度に対して初速度を加算する方法もあります。この場合は現在の速度の影響が強くなるため、ボタンを入力するタイミングが頂点の前か後かによって、2段目のジャンプの高さが大きく変わります。ゲームの内容に応じて、どちらでも操作性がよくなる方法を選ぶとよいでしょう。

## プログラム Program

List 2-3は2段ジャンプのプログラムです。定数を変更すれば、ジャンプの段数を変えることができます。また、2段ジャンプ時に速度に初速度を加算するように処理を変えて、動きの違いを比べてみてください。

**List 2-3** 2段ジャンプ(CDoubleJumpManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.15f;

    // ジャンプの初速度
    float jump_speed=-0.4f;

    // ジャンプの加速度
    float jump_accel=0.02f;

    // 地面にキャラクターがいるときのY座標
    // 着地の判定に使う
    float ground_y=MAX_Y-2;

    // 2段ジャンプが入力可能な速度の範囲
    // Y方向の速度の絶対値がこの値よりも小さいときに、
    // ボタンを入力すると2段ジャンプができる
    // この値を小さくすると2段ジャンプできる範囲が狭くなり、
    // 大きくすると範囲が広くなる
    float double_jump_vy=0.1f;

    // ジャンプの段数の上限
    // ここを3や4にすると、3段ジャンプや4段ジャンプが可能になる
    int jump_count=2;

    // 通常状態の処理
    if (JumpCount==0) {

        // レバーの入力にしたがってX方向の速度を設定する
        // VXはキャラクターのX方向の速度
        VX=0;
        if (is->Left) VX=-speed;
        if (is->Right) VX=speed;

        // ボタンを押したらジャンプ状態に移行する
        // ジャンプの段数と初速度を設定する
        // JumpCountはジャンプの段数
        // VYはキャラクターのY方向の速度
        if (is->Button[0]) {
            JumpCount=1;
            VY=jump_speed;
        }
    } else

    // ジャンプ状態の処理
    {
```

**List 2-3**

```
		// Y方向の速度に加速度を加える
		VY+=jump_accel;

		// Y座標の更新
		Y+=VY;

		// 着地の判定
		// キャラクターが落下中(Y方向の速度が正の値)で、
		// かつ地面にキャラクターがいるときのY座標に達していたら、
		// 着地したと判定する
		// ジャンプの段数を0にして通常状態に戻り、
		// Y座標を地面にいるときの座標に設定する
		if (VY>0 && Y>=ground_y) {
			JumpCount=0;
			Y=ground_y;
		}

		// 2段ジャンプの処理
		// 以下の条件をすべて満たしたときに2段ジャンプになる
		// ・ジャンプの頂点付近にいる
		//   (Y方向の速度の絶対値が一定値未満)
		// ・ボタンを押した瞬間である
		//   (直前にボタンを放していて、現在はボタンを押している)
		// ・ジャンプの段数が規定回数未満である
		// PrevButtonは直前のボタンの状態、
		// is->Button[0]は現在のボタンの状態を表す
		if (
			abs(VY)<double_jump_vy &&
			!PrevButton && is->Button[0] &&
			JumpCount<jump_count
		) {
			// Y方向の速度に初速度を設定することによって、
			// 2段ジャンプを行う
			// また、ジャンプの段数を増加させる
			// なお、速度に初速度を加算する方法もある
			// (VY+=jump_speed)
			VY=jump_speed;
			JumpCount++;
		}
	}

	// X座標を更新し、画面からはみ出さないように補正する
	X+=VX;
	if (X<0) X=0;
	if (X>MAX_X-1) X=MAX_X-1;

	// 現在のボタンの状態を、直前のボタンの状態として保存する
	PrevButton=is->Button[0];

	// X方向の速度に応じて、キャラクターを傾けて表示する
```

```
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「DOUBLE JUMP」は2段ジャンプのサンプルです。左右のレバーでキャラクターが移動し、ボタンでジャンプします。キャラクターが静止している状態でボタンを押せば垂直に、左右に移動中にボタンを押せば放物線状にジャンプします。ジャンプの頂上付近でタイミングよくボタンを押せば、2段ジャンプを行います。

**DOUBLE JUMP** → p. 393

## 三角跳び

壁に向かってジャンプし、壁に接触したときにジャンプボタンを押すと、キャラクターが壁を蹴ってジャンプするアクションです。壁を蹴って反対側に跳ぶ際の軌跡が三角形状になることから、「三角跳び」と呼ばれています。トリッキーな動きができるので、アクションゲームのほかに、格闘ゲームにもよく採用されています。なお、壁で三角跳びを行う以外に、画面の端を壁に見立てて三角跳びができるゲームも多いようです。

三角跳びを行うには、まず壁に向かってジャンプすることから始めます (Fig. 2-28)。レバーを壁の方向に入力し、壁に向かって走りながらジャンプボタンを押します。

壁に向かってジャンプしたら、一度ボタンを放します。そして、キャラクターが壁に接触し

**Fig. 2-28** 壁に向かってジャンプする

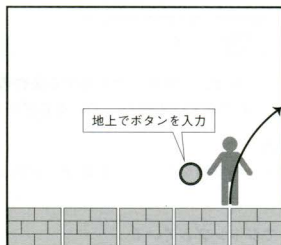



Fig. 2-29 三角跳びの入力

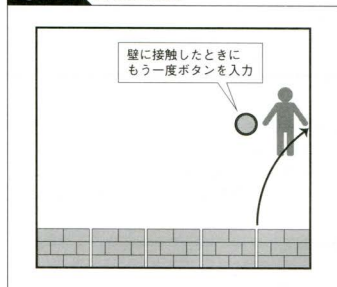
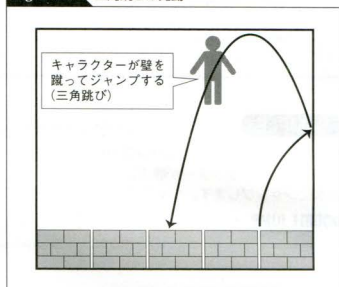



Fig. 2-30 三角跳びの発動

たときに、もう一度ボタンを入力します(Fig. 2-29)。

うまく壁際でボタンを押すと、三角跳びが発動します(Fig. 2-30)。キャラクターが壁を蹴ってジャンプし、最初のジャンプとは逆の左右方向へ、より高くジャンプします。

三角跳びは格闘系アクションゲームや格闘ゲームに多く採用されています。例えば「イーアルカンフー」や「ファイナルファイト」「サムライスピリッツ」「龍虎の拳」といったゲームに、三角跳びが使われています。三角跳びの入力方法はゲームによって多少違っていて、ボタンで入力する場合もあれば、レバーで入力する場合もあります。あるいは「イーアルカンフー」のように、何も入力しなくても、壁に向かってジャンプすれば自動的に三角跳びになるゲームもあります。

格闘系以外では、「スーパーマリオブラザーズ」にも三角跳びがあります。初代の「スーパーマリオブラザーズ」では、壁を構成しているブロックのすき間に足を引っかけてジャンプするという、隠れたテクニックでした。続編の「スーパーマリオブラザーズ2」では、普通に使えるテクニックとして採用されました。

## アルゴリズム Algorithm

三角跳びを実現する際のポイントは、三角跳びが発動する条件の判定です(Fig. 2-31)。三角跳びが発動するには、次の3つの条件をすべて満たしている必要があります。

・壁とキャラクターの距離が近い

壁または画面端とキャラクターの座標を比べて、距離が一定値より小さいかどうかを調べます。

・キャラクターが壁に向かってジャンプしている

キャラクターがジャンプ状態であり、なおかつX方向の速度が壁に近づく向きかどうかを調べます。

・ボタンを入力した瞬間である

直前と現在のボタンの状態を調べて、ボタンを一度放してから入力したかどうかを調べます。

三角跳びが発動したら、キャラクターの速度を変更することによって、三角跳びの動きを作り出します(Fig. 2-32)。例えば、次のように速度を変更します。

・X方向の速度を反転させる

左右の進行方向を逆にすることで、壁で跳ね返る様子を表現します。

**Fig. 2-31** 三角跳びが発動する条件

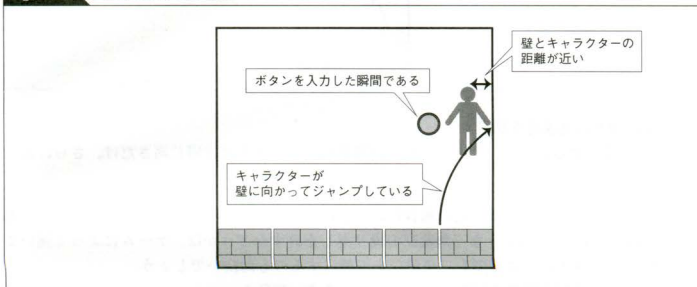



**Fig. 2-32** 三角跳びの速度制御

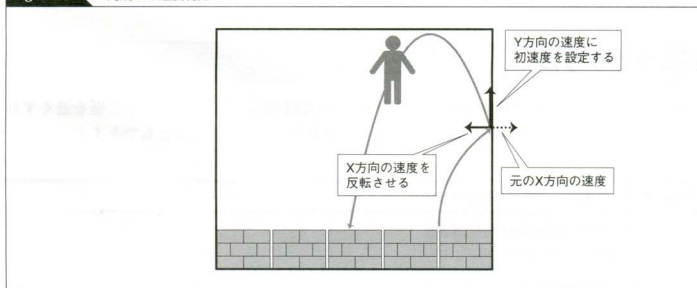
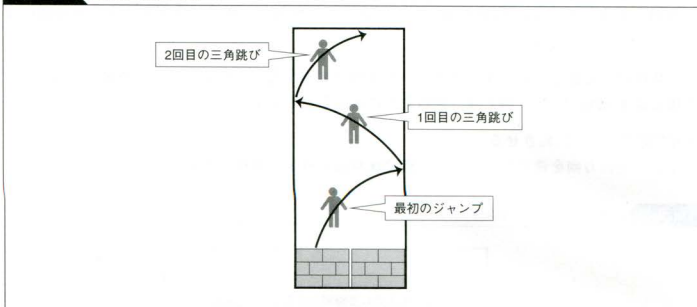



Fig. 2-33 三角跳びを繰り返す

・Y方向の速度に初速度を設定する

初速度を設定することで、キャラクターは最初のジャンプとほぼ同じ高さだけ、さらに跳び上がります。

ゲームによっては、壁と壁が近い場合に、三角跳びを何度も繰り返すことができるものもあります（Fig. 2-33）。このような三角跳びの繰り返しを許すかどうかは、ゲームによって違います。壁があるかぎり、どこまでも三角跳びを可能にするのも面白いでしょう。

一方で、三角跳びの回数を制限することもできます。回数を制限する場合には、三角跳びを実行した回数を数えておき、三角跳びを入力するたびに、回数が上限に達していないかどうかを調べます。

## プログラム Program

List 2-4は三角跳びのプログラムです。ジャンプの回数制限を増やして、壁と壁を近くすれば、左右の壁を使って三角跳びで上がっていくプログラムに改造することもできます。

**List 2-4** 三角跳び（CTriangleJumpManクラス）

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.15f;

    // ジャンプの初速度
    float jump_speed=-0.4f;
```

```
// ジャンプの加速度
float jump_accel=0.02f;

// 地面にキャラクターがいるときのY座標
// 着地の判定に使う
float ground_y=MAX_Y-2;

// 三角跳びが入力可能な距離の範囲
// 壁とキャラクターの距離がこの値よりも近いときに、
// ボタンを入力すると三角跳びができる
// この値を小さくすると三角跳びができる範囲が狭くなり、
// 大きくすると範囲が広くなる
float wall_distance=0.1f;

// ジャンプの回数の上限
// ここを3以上にすると、三角跳びを2回以上繰り返し行える
// ただし、壁際でないと三角跳びはできないので、
// 繰り返し行わせるには壁と壁を近く設置することも必要
int jump_count=2;

// 通常状態の処理
if (JumpCount==0) {

    // レバーの入力にしたがってX方向の速度を設定する
    // VXはキャラクターのX方向の速度
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // ボタンを押したらジャンプ状態に移行する
    // ジャンプの回数と初速度を設定する
    // JumpCountはジャンプの回数
    // VYはキャラクターのY方向の速度
    if (is->Button[0]) {
        JumpCount=1;
        VY=jump_speed;
    }
} else

// ジャンプ状態の処理
{
    // Y方向の速度に加速度を加える
    VY+=jump_accel;

    // Y座標の更新
    Y+=VY;

    // 着地の判定
    // キャラクターが落下中(Y方向の速度が正の値)で、
```

**List 2-4**

```
		// かつ地面にキャラクターがいるときのY座標に達していたら、
		// 着地したと判定する
		// ジャンプの回数を0にして通常状態に戻り、
		// Y座標を地面にいるときの座標に設定する
		if (VY>0 && Y>=ground_y) {
			JumpCount=0;
			Y=ground_y;
		}

		// 三角跳びの処理
		// 以下の条件をすべて満たしたときに三角跳びになる
		// ・壁に向かって進んでいて、かつ壁との距離が近い
		//  (左に進んでいて左の壁に近いか、右に進んでいて右の壁に近い)
		// ・ボタンを押した瞬間である
		//  (直前にボタンを放していて、現在はボタンを押している)
		// ・ジャンプの回数が規定回数未満である
		// PrevButtonは直前のボタンの状態、
		// is->Button[0]は現在のボタンの状態を表す
		if (
			(VX<0 && X<wall_distance || VX>0 && MAX_X-1-X<wall_distance) &&
			!PrevButton && is->Button[0] &&
			JumpCount<jump_count
		) {
			// X方向の速度を反転させ、
			// Y方向の速度に初速度を加えることによって三角跳びを行う
			// また、ジャンプの回数を数える
			VX=-VX;
			VY+=jump_speed;
			JumpCount++;
		}
	}

	// X座標を更新し、画面からはみ出さないように補正する
	X+=VX;
	if (X<0) X=0;
	if (X>MAX_X-1) X=MAX_X-1;

	// 現在のボタンの状態を、直前のボタンの状態として保存する
	PrevButton=is->Button[0];

	// X方向の速度に応じて、キャラクターを傾けて表示する
	Angle=VX/speed*0.1f;

	return true;
}
```

SAMPLE

「TRIANGLE JUMP」は三角跳びのサンプルです。左右のレバーでキャラクターが移動し、ジャンプボタンでジャンプします。キャラクターが静止している状態でボタンを押せば垂直に、左右に移動中にボタンを押せば放物線状にジャンプします。ジャンプ中に左右の壁に接している状態でボタンを押せば、三角跳びを行います。

**TRIANGLE JUMP** → p. 393

## 飛び降り

キャラクターが床に乗っているときに、レバーを下に入れながらジャンプすると(ジャンプボタンを押すと)、キャラクターが下に飛び降りるアクションです。床が何層も重なっているようなステージで、素早く下の層に飛び降りるときに使います。

飛び降りの前に、まずジャンプで床に乗る動作について考えましょう。最初に、ボタンを押してジャンプを開始します(Fig. 2-34)。

ジャンプしたキャラクターは、床を通過して上昇します(Fig. 2-35)。このように床を通過できるかどうかはゲームによって違いますが、飛び降りを採用した多くのゲームでは、ジャンプで床を通過することができます。

床の上までジャンプして、床に乗ることに成功すると、キャラクターは着地します(Fig. 2-36)。もし上の床に届かなかった場合は、着地できる床が出現するまで、キャラクターは落下していきます。

次に、飛び降りの動きについて考えましょう。飛び降りの操作はゲームによって違いますが、多くのゲームではレバーを下や斜め下に入れながらジャンプボタンを押すと、飛び降りができます(Fig. 2-37)。

飛び降りたキャラクターは、床を通過して落下します(Fig. 2-38)。飛び降りを採用したほと

**Fig. 2-34** ジャンプの開始

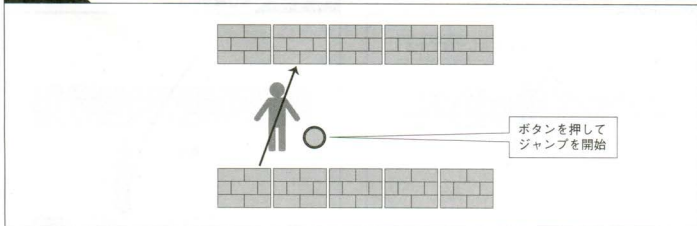



**Fig. 2-35** 床を通過して上昇する

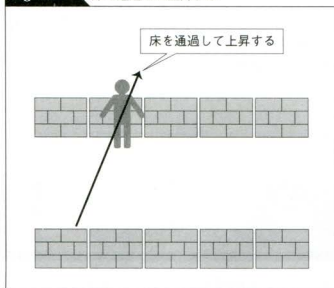



**Fig. 2-36** ジャンプの着地

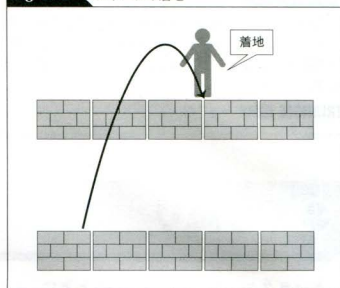



**Fig. 2-37** 飛び降りの開始

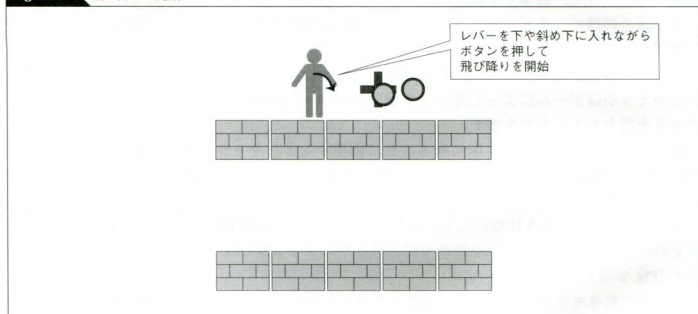



**Fig. 2-38** 床を通過して落下する

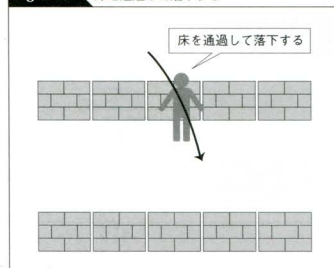
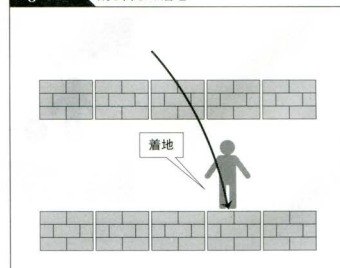



**Fig. 2-39** 飛び降りの着地

んどのゲームでは、このように床を通過することができます。

下の床に到達すると、キャラクターは着地します(Fig. 2-39)。これで飛び降りは完了です。なお、着地できる床が出現するまで、キャラクターは落下を続けます。

## 床の端から落ちる

飛び降りに関連して、キャラクターが床の端から落ちた場合についても考えてみましょう。左右に移動するキャラクターが床を踏み外すと、落下が始まります。床を踏み外したキャラクターは、重力に引かれて落下します(Fig. 2-40)。重力があるため、キャラクターの落下スピードはだんだん大きくなっていきます。

着地できる床に到達すると、キャラクターは着地します(Fig. 2-41)。着地に関しては飛び降りと同様です。飛び降りはレバーとジャンプボタンを同時に入力することによって始まり、踏み外しは足下に床がなくなったときに始まる、という点だけが異なります。

Fig. 2-40 床の端から落ちる

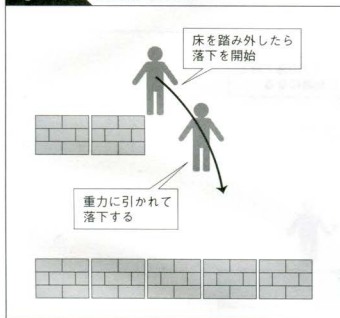
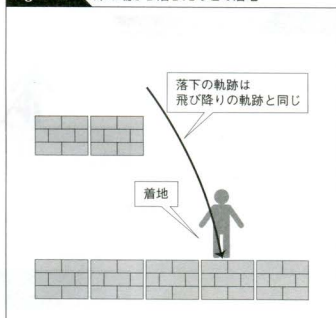



Fig. 2-41 床の端から落ちたあとの着地

飛び降りを採用しているゲームは数多くあります。例えば「魂斗羅」では、レバーを下に入れてジャンプボタンを押すと、床から飛び降りることができます。「忍者くん」の場合には、レバーを左右に入れながらボタンを押すとジャンプになり、レバーを入れずにボタンを押すと飛び降りになります。

「忍者くん」では、ジャンプや飛び降りが攻撃手段にもなっています。階層が違う床にいる敵に向かってジャンプや飛び降りで体当たりすると、敵を気絶させることができます。そのため、最初にジャンプや飛び降りで敵を気絶させておき、手裏剣でとどめを刺すというアクションが楽しめます。

また「ソンソン」にも飛び降りが採用されています。このゲームは何層かの床が重なったステージ構成になっており、レバーを上に入れると上の床にジャンプし、レバーを下に入れると

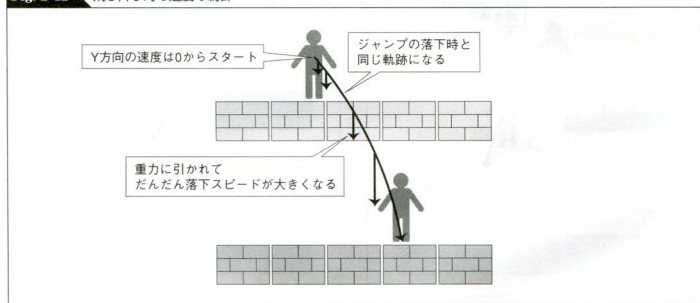
下の床に飛び降りることができます。

床を踏み外したときに落下するアクションは、飛び降りを採用していないゲームでも広く使われています。飛び降りと踏み外しの処理には共通部分が多いので、ここではまとめて解説することにしました。

## アルゴリズム Algorithm

飛び降りを実現する際の最初のポイントは、飛び降りが発動したときの速度の制御です(Fig. 2-42)。飛び降りた瞬間には、Y方向の速度を0にしておきます。飛び降りが進行するにつれて、速度に重力の加速度が加わるため、落下スピードはだんだん大きくなります。結果として、飛び降りの軌跡は、ジャンプの落下時と同じ軌跡(放物線)になります。

キャラクターを一定速度で落下させるだけでもよいのですが、重力が働いているかのように表現した方が落ちていく雰囲気が出ます。特にキャラクターの落下距離が長い場合には有効です。

**Fig. 2-42** 飛び降り時の速度の制御

### 着地の処理

次のポイントは着地の処理です。キャラクターが着地したかどうかを判定するには、キャラクターの直下に床があるかどうかを判定する必要があります。これは一種の当たり判定処理です。

なお、着地の判定処理を行うのは、キャラクターが落下している間だけです。キャラクターが上昇している間には、着地の判定処理は行いません。これは、上昇中にはキャラクターに床を突き抜けて進んでほしいので、床との当たり判定処理を行うと不都合が生じるからです。

まずX方向に関しては、キャラクターと床のX座標の差分が、一定範囲内にあるかどうかを調べます(Fig. 2-43)。例えば、キャラクターの座標を(x, y)、床の座標を(fx, fy)、これら2つ

のX座標の差分の最大値をfloor_max_xとすると、キャラクターが床に乗るための条件は、

```
fx-x<floor_max_x && x-fx<floor_max_x
```

となります。絶対値を返すabs関数を使えば、

```
abs(fx-x)<floor_max_x
```

と書くこともできます。

floor_max_xの値を小さくすると床から落ちやすくなり、大きくすると落ちにくくなります。

**Fig. 2-43** X方向に関する着地の判定

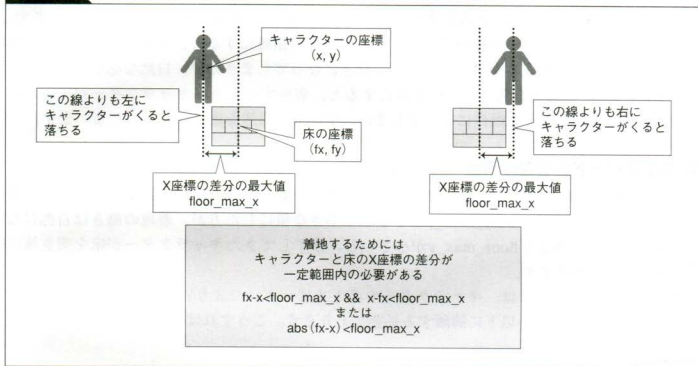
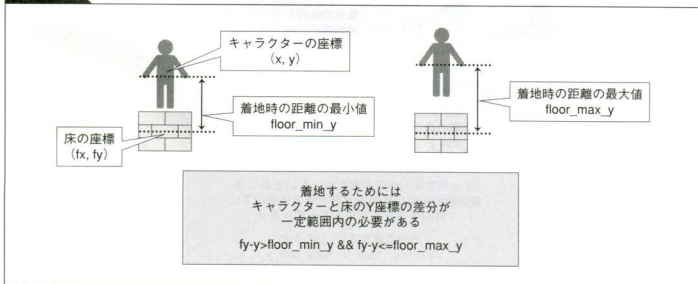



**Fig. 2-44** Y方向に関する着地の判定

一般に、多少床から落ちにくくしておいた方が、うっかり床から落ちてしまうことが少なくなって、遊びやすいゲームになります。

続いてY方向に関しても、キャラクターと床のY座標の差分が一定範囲内にあるかどうかを調べます(Fig. 2-44)。例えば、キャラクターの座標を(x, y)、床の座標を(fx, fy)、Y座標の差分の最小値と最大値をそれぞれfloor_min_yとfloor_max_yとすると、キャラクターが床に乗るための条件は、

```
fy-y>floor_min_y && fy-y<=floor_max_y
```

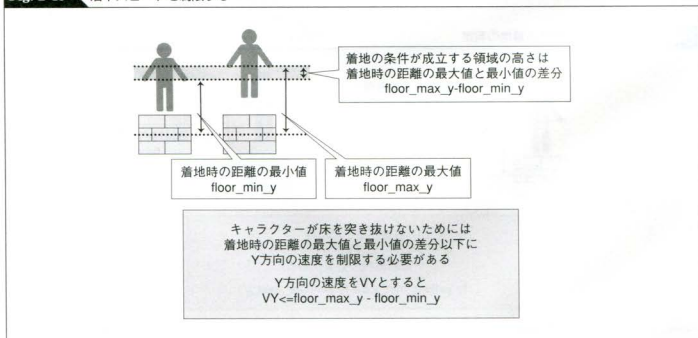
となります。比較演算子の片方が「>」で、もう片方が「<=」になっている理由は後述します。

floor_min_yはキャラクターが床にちょうど乗っている距離に、floor_max_yはキャラクターが床の少しだけ上の空中にいるときの距離に設定すると、着地の動作が自然になります。floor_min_yをキャラクターが床面よりも下にいるときの距離にすると、キャラクターが少し床にめり込んだあとに床面まで出てくるような動きになってしまうので、自然な感じになりません。また、floor_max_yをあまり遠い距離にすると、着地するときにキャラクターが床面に磁石で吸いつけられたような動作になってしまい、やはり不自然な感じになってしまいます。

## 落下スピードと位置の調整

floor_max_yの値を決めるときには、キャラクターの落下スピードとの関係を考慮する必要があります(Fig. 2-45)。floor_max_yはなるべく小さな値にした方が、着地の動きは自然になります。しかし、あまりfloor_max_yが小さいと、落下してきたキャラクターが床を突き抜けてしまうことがあります。

突き抜けを防ぐためには、キャラクターの落下スピード、つまりY方向の速度の大きさを、floor_max_y - floor_min_y以下に制限する必要があります。こうすれば、

**Fig. 2-45** 落下スピードを制限する

```
fy-y>floor_min_y && fy-y<=floor_max_y
```

という前述の着地の条件が成立する領域に、落下する途中にキャラクターが必ず1フレームは存在することになります。そのため、床を突き抜けることがなくなります。

もしも落下スピードを制限しないと、キャラクターが移動する1フレームの間に、着地条件が成立する領域を通り過ぎてしまう可能性があります。つまり、運が悪いと床を突き抜けてしまうということです。

さて、着地したあとには、床とキャラクターの距離が着地時の最小の距離floor_min_yになるように、キャラクターのY座標を調整します(Fig. 2-46)。キャラクターの座標が(x, y)、床の座標が(fx, fy)のときには、

```
y=fy-floor_min_y
```

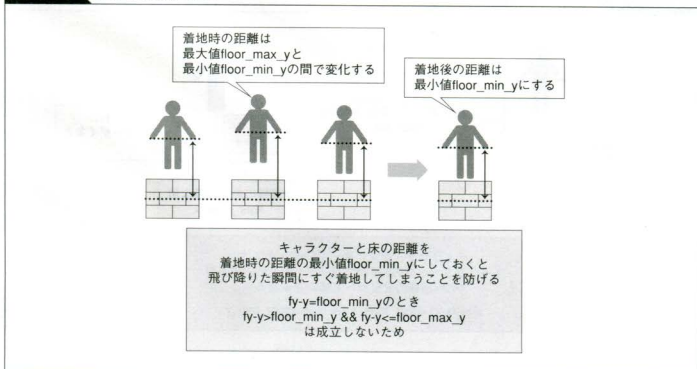
となります。

着地した瞬間のキャラクターのY座標は、状況によって微妙に異なります。Y座標を調整するのは、状況による違いを吸収して、キャラクターがぴったりと床面に乗った状態に揃えるためです。

床とキャラクターの距離をfloor_min_yに合わせるのには理由があります。これは、飛び降りた直後に、飛び降りた床に着地したと判定されないための処置です。前述の、

```
fy-y>floor_min_y && fy-y<=floor_max_y
```

という着地を判定する条件式では、1つ目の比較演算子が「>」になっています。飛び降りた瞬間には、

**Fig. 2-46** 着地後のY座標の調整

```
y=fy-floor_min_y
```

であり、これはすなわち、

```
fy-y==floor_min_y
```

ということです。つまり、飛び降りた瞬間にはキャラクターは床の上にいますが、着地の条件は成立しないようにしてあります。このように着地の条件式やキャラクターのY座標を工夫しておかないと、飛び降りた瞬間に同じ床に着地してしまい、うまく飛び降りられないといったバグが発生する可能性があるので注意が必要です。

### 床の端の判定処理

キャラクターが床の端から落下するかどうかの判定処理についても考えてみましょう（Fig. 2-47）。これは着地の判定処理とよく似ています。キャラクターが床に乗っているために必要な条件を見てみると、まずX方向については、

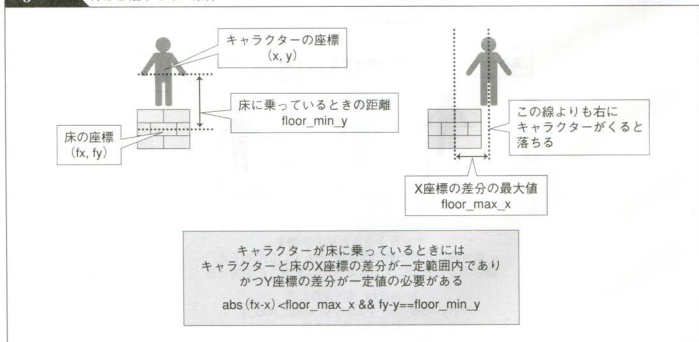
```
fx-x<floor_max_x && x-fx<floor_max_x
```

または、

```
abs(fx-x)<floor_max_x
```

のように、キャラクターと床のX座標の差分が一定範囲内である必要があります。Y方向については、

```
fy-y==floor_min_y
```

Fig. 2-47　床から落下しない条件

のように、キャラクターと床のY座標の差分が一定値でなければなりません。

X方向とY方向の条件がともに成立しているときには、キャラクターが床に乗っているということです。ステージのすべての床について判定して、キャラクターがいずれかの床に乗っていたらキャラクターは落下しません。どの床にも乗っていないときには、キャラクターの足下に床がないということなので落下を開始します。

## プログラム Program

List 2-5は飛び降りのプログラムです。少し長くなりましたが、床の端から外れたときに落下する処理が不要な場合には、該当する部分を省略すれば簡単になります。

なお、このプログラムではキャラクターとすべての床について着地の判定処理を行いますが、格子状に区切られたマップに床を配置しておき、キャラクターのすぐ近くにある床についてだけ判定処理を行う方法もあります。この方法は判定処理が少なくなることが利点ですが、床や壁を格子状に整然と配列する必要があるため、マップの作り方には工夫がいります。一方、本書のプログラムの方法では判定処理が多くなりますが、格子状に置くことにとらわれず、床や壁を自由な座標に配置することができます。ゲームを動かすマシンの性能などに応じて、適切な方法を選ぶとよいでしょう。最近のPCの場合には、本書の方法でもあまり問題はありません。

**List 2-5** 飛び降り(CJumpOffManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.15f;

    // ジャンプの初速度
    float jump_speed=-0.4f;

    // ジャンプの加速度
    float jump_accel=0.02f;

    // 一番低い地面にキャラクターがいるときのY座標
    // 飛び降りができるかどうかの判定に使う
    float bottom_y=MAX_Y-2;

    // キャラクターが床に乗っているかどうかを判定するときの、
    // X座標の差分の最大値
    float floor_max_x=0.8f;

    // キャラクターが床に乗っているかどうかを判定するときの、
    // Y座標の差分の最小値と最大値
    float floor_min_y=1.0f;
```

**List 2-5**

```
float floor_max_y=1.4f;

// 通常状態の処理
if (!Jump) {

    // レバーの入力にしたがってX方向の速度を設定する
    // VXはキャラクターのX方向の速度
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // レバーを下に入れずにボタンを押したら、
    // ジャンプ状態に移行する
    // ジャンプ状態のフラグをtrueにして、初速度を設定する
    // Jumpはジャンプ状態かどうかを表すフラグ
    // VYはキャラクターのY方向の速度
    if (is->Button[0] && !is->Down) {
        Jump=true;
        VY=jump_speed;
    }

    // レバーを下に入れながらボタンを押したら、
    // 飛び降り状態に移行する
    // ただし、一番低い地面にいるときには飛び降りない
    // ジャンプ状態のフラグをtrueにして、速度を0にする
    // キャラクターは重力に引かれて落下する
    if (is->Button[0] && is->Down && Y<bottom_y) {
        Jump=true;
        VY=0;
    }

    // 床から外れたときに落下する処理を行わない場合には、
    // ここからの処理は不要

    // キャラクターが床から外れたときに落下する処理
    // キャラクターの足下に床があるかどうかを調べる
    bool on_floor=false;
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();

        // 床とキャラクターのX方向の距離が一定値未満で、
        // かつ床のY座標がキャラクターのY座標のすぐ下ならば、
        // キャラクターが床に乗っていると判定する
        if (
            mover->Type==1 &&
            abs(mover->X-X)<floor_max_x &&
            mover->Y-Y==floor_min_y
        ) {
            on_floor=true;
            break;
```

```
			}
		}

		// キャラクターが一番低い地面にはおらず、
		// かつ床に乗っていないときには、落下する
		// 飛び降り状態と同じように、
		// ジャンプ状態のフラグをtrueにして、速度を0にする
		// キャラクターは重力に引かれて落下する
		if (Y<bottom_y && !on_floor) {
			Jump=true;
			VY=0;
		}

		// 床から外れたときに落下する処理を行わない場合には、
		// ここまでの処理は不要

	} else

	// ジャンプ状態の処理
	{
		// Y方向の速度に加速度を加える
		VY+=jump_accel;

		// 落下スピードが一定値以上にならないように補正する
		// 落下スピードが大きくなりすぎると、
		// 床とキャラクターの当たり判定処理がうまくいかなくなり、
		// キャラクターが床を突き抜けてしまうため
		if (VY>floor_max_y-floor_min_y) {
			VY=floor_max_y-floor_min_y;
		}

		// Y座標の更新
		Y+=VY;

		// 着地の処理
		// 床とキャラクターの当たり判定処理を行い、
		// キャラクターの足下が床ならば着地する
		for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
			CMover* mover=(CMover*)i.Next();

			// キャラクターが落下中で、
			// 床とキャラクターのX方向の距離が一定値未満であり、
			// かつ床のY座標がキャラクターのY座標のすぐ下ならば、
			// キャラクターが着地したと判定する
			// キャラクターが床を突き抜けないようにするために、
			// Y座標の範囲は少し広めにしてある
			if (
				VY>0 &&
				mover->Type==1 &&
				abs(mover->X-X)<floor_max_x &&
```

**List 2-5**

```
                    mover->Y-Y>floor_min_y &&
                    mover->Y-Y<=floor_max_y
                ) {
                    // ジャンプ状態のフラグをfalseにして、
                    // キャラクターがちょうど床面に乗るように、
                    // Y座標を設定する
                    Jump=false;
                    Y=mover->Y-floor_min_y;
                    break;
                }
            }
        }

        // X座標を更新し、画面からはみ出さないように補正する
        X+=VX;
        if (X<0) X=0;
        if (X>MAX_X-1) X=MAX_X-1;

        // X方向の速度に応じて、キャラクターを傾けて表示する
        Angle=VX/speed*0.1f;

        return true;
    }
```

**SAMPLE**

「JUMP OFF」は飛び降りのサンプルです。左右のレバーでキャラクターが移動し、ボタンでジャンプします。ジャンプで上のフロアに飛び上がることができます。下方向のレバーとボタンを同時に入力すると、下のフロアに飛び降ります。最下層のフロアでは飛び降りはできません。また、フロアの左右の端からはみ出すと下に落下します。

**JUMP OFF** → p. 393

# ジャンプ飛行

空中でボタンを押すと、キャラクターが空中でジャンプして上昇するアクションです。ボタンを連続して入力することによって、空中を自由に飛行することができます。

最初は地上でジャンプボタンを押すことから始まります(Fig. 2-48)。ボタンを押すと、キャラクターは普通にジャンプします。

次に空中で再びボタンを押すと、キャラクターは空中でジャンプの動作を行います(Fig. 2-49)。地上でのジャンプと同じ放物線を描きながら、キャラクターはさらに上昇します。

「2段ジャンプ(→ p. 72)」と違うのは、ジャンプの頂点以外でもボタンの入力を受け付ける

ことです (Fig. 2-50)。キャラクターが上昇していても、あるいは落下していても、ボタンを押せばいつでも空中でジャンプします。

さらに、レバーを入力することによって、空中でも左右へ自由に移動することができます。ジャンプと左右移動を組み合わせれば、空中を自由に飛行することが可能です (Fig. 2-51)。

ジャンプ飛行を採用しているゲームには、例えば「パックランド」があります。このゲームでは、通常時のキャラクターは固定長ジャンプ (→ p. 60) を行いますが、特定の条件を満たす

**Fig. 2-48** 地上でジャンプする

地上でボタンを押すと
ジャンプする

**Fig. 2-49** 空中でジャンプする

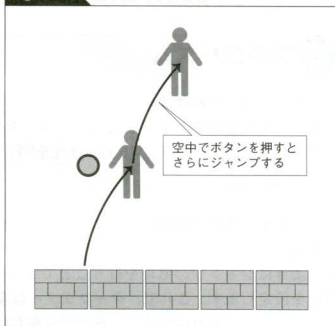



**Fig. 2-50** いつでもジャンプができる

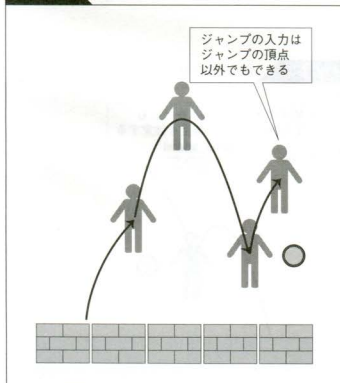
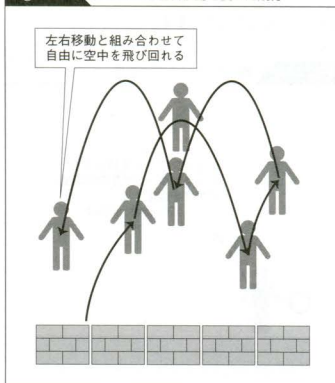



**Fig. 2-51** ジャンプと左右移動を使った飛行

とジャンプ飛行ができるようになります。通常時では越えるのが難しかった谷間や池などが、ジャンプ飛行を使うと簡単に越えられるので、とても爽快感があります。ジャンプの特性を変えることによって、似たステージでまったく違ったゲーム性を出しているという点で、非常に上手なアクションの使い方だといえます。

ボタンで飛行するゲームには、「バルーンファイト」や「Joust」などがあります。これらのゲームの場合、空中でボタンを連続入力することで飛行するという点は同じですが、「パックランド」のように空中でジャンプの動作を行うわけではありません。どちらかといえば、泳ぐアクション(→ p. 37)に似ています。

## アルゴリズム Algorithm

ジャンプ飛行の処理は、固定長ジャンプの処理をアレンジしたものです。まず、地上にいるときにジャンプボタンを押した場合には、キャラクターのY方向の速度に、ジャンプの初速度を設定します(Fig. 2-52)。Y方向の速度をVY、初速度をjump_speedとすると、

```
VY=jump_speed
```

となります。これは固定長ジャンプとまったく同じ処理です。

固定長ジャンプと違うのは、空中でボタンを押した場合にも、地上でボタンを押した場合と同じようにジャンプの処理を行うことです(Fig. 2-53)。ボタンを押したら、Y方向の速度に初速度を設定します。これで、キャラクターは空中でも地上と同じようにジャンプします。

また、レバー入力に応じて、空中でも左右に移動できるようにします(Fig. 2-54)。地上でレバーを入力したときと同じように、レバーの方向に応じてキャラクターを左右に動かします。

キャラクターが地上に降りたら、着地の処理を行います(Fig. 2-55)。落下を止めるために、Y方向の速度を0にします。

**Fig. 2-52** 地上でのジャンプの処理

**Fig. 2-53** 空中でのジャンプの処理

**Fig. 2-54** 空中での左右移動

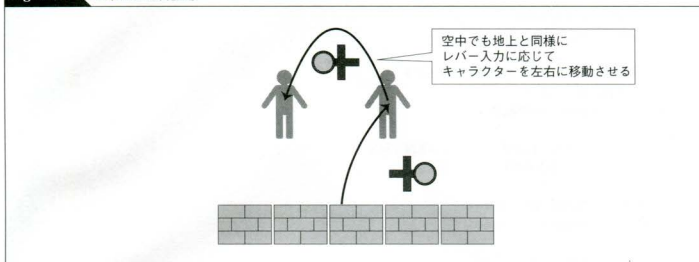
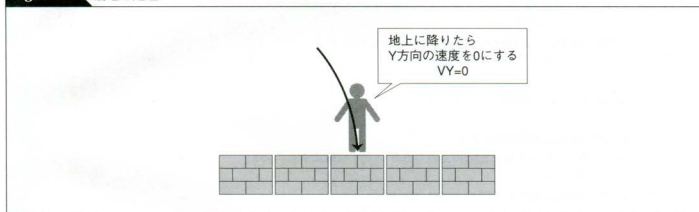



**Fig. 2-55** 着地の処理

## プログラム Program

List 2-6はジャンプ飛行のプログラムです。ジャンプ飛行では、地上でも空中でも同じようにジャンプや左右移動ができるので、通常状態とジャンプ状態を区別していません。固定長ジャンプではジャンプ状態かどうかを区別するためのフラグを用意していましたが、そういったフラグは不要です。固定長ジャンプのプログラム（→ p. 64）と比べてみてください。

**List 2-6** ジャンプ飛行（CJumpFlyManクラス）

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.15f;

    // ジャンプの初速度
    float jump_speed=-0.3f;
```

List 2-6

```
	// ジャンプ中の加速度
	float jump_accel=0.03f;

	// 地面にキャラクターがいるときのY座標
	// 着地の判定に使う
	float ground_y=MAX_Y-2;

	// レバーの入力にしたがってX方向の速度を設定する
	// VXはキャラクターのX方向の速度
	VX=0;
	if (is->Left) VX=-speed;
	if (is->Right) VX=speed;

	// ボタンを押したら、
	// Y方向の速度にジャンプの初速度を設定する
	// VYはキャラクターのY方向の速度
	if (is->Button[0]) VY=jump_speed;

	// X座標を更新し、画面からはみ出さないように補正する
	X+=VX;
	if (X<0) X=0;
	if (X>MAX_X-1) X=MAX_X-1;

	// Y方向の速度に加速度を加える
	// また、落下スピードが一定値以上にならないように補正する
	VY+=jump_accel;
	if (VY>-jump_speed) VY=-jump_speed;

	// Y座標を更新し、画面からはみ出さないように補正する
	Y+=VY;
	if (Y<0) Y=0;

	// キャラクターが落下中(Y方向の速度が正の値)で、
	// かつ地面にキャラクターがいるときのY座標に達していたら、
	// 着地したと判定する
	// Y方向の速度を0にして、
	// Y座標を地面にいるときの座標に設定する
	if (VY>0 && Y>=ground_y) {
		VY=0;
		Y=ground_y;
	}

	// X方向の速度に応じて、キャラクターを傾けて表示する
	Angle=VX/speed*0.1f;

	return true;
}
```

SAMPLE

「UNIVERSAL JUMP」はジャンプ飛行のサンプルです。左右のレバーでキャラクターが移動し、ボタンでジャンプします。ジャンプ中に再度ボタンを押すことで、何度でもジャンプを繰り返します。また、ジャンプ中はレバー左右でキャラクターを操作することができます。

**UNIVERSAL JUMP** → p. 393

# ジャンプ角度調整

ボタンを押してジャンプするときに、ボタンを押し下げたままにすることによって、ジャンプの角度を調整するアクションです。ボタンを放したときの角度によって、ジャンプの飛距離が変わります。

最初は、キャラクターは地上にいます (Fig. 2-56)。ゲームによってレバーを使うかボタンを使うかは違いますが、いずれにしてもキャラクターは地上を移動することができます。

ジャンプボタンを押すと、ジャンプの体勢に入ります (Fig. 2-57)。ボタンを押しっぱなしにすると、キャラクターはまだジャンプせず、足もその場に止めたままで、ジャンプの角度だけが変わります (Fig. 2-58)。ここでは、キャラクターを傾けてジャンプの角度を表現しましたが、ゲームによってはゲージなどで表示する場合もあります。

ジャンプしたい角度になったときにボタンを放すと、その角度でキャラクターがジャンプします (Fig. 2-59)。ジャンプの飛距離は、ジャンプの体勢に入る前の移動速度と、ジャンプの角度から決まります。

ジャンプ角度調整のアクションを採用したゲームの例としては、「ハイパーオリンピック」があります。このゲームでは幅跳びや高跳びといった競技を行いますが、ジャンプのときにボタンを押しっぱなしにすることによって、ジャンプの角度を調整することができます。ジャンプの角度によって飛距離が変わるので、記録を出すためには適切な角度を覚えておき、うまく角度を合わせる必要があります。

**Fig. 2-56** 地上を移動する

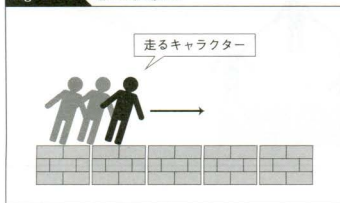



**Fig. 2-57** ジャンプの体勢に入る

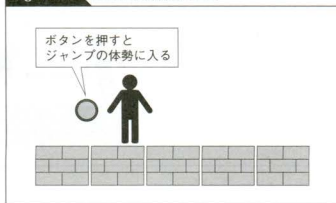



Fig. 2-58 ボタンを押しっぱなしにしてジャンプの角度を調整する

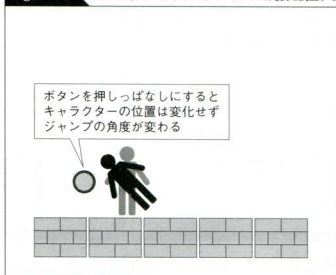
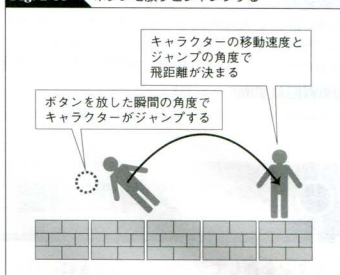
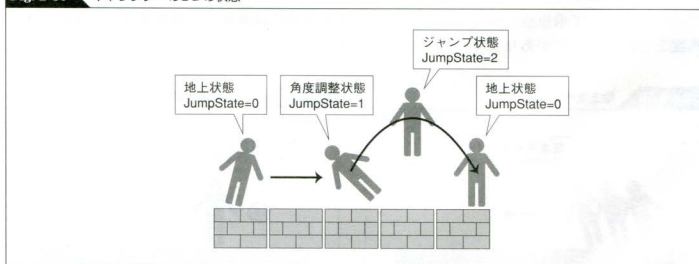



Fig. 2-59 ボタンを放すとジャンプする

そのうえ「ハイパーオリンピック」では、ジャンプ前の助走でスピードを上げるために、ボタンを連打(→ p. 27)しなければいけません。猛烈な勢いでボタンを連打した直後に、今度はジャンプボタンを適切な長さだけ押して角度を調整する必要があるので、プレイヤーは大変です。単にボタンを速く連打すればよいわけではない点が、このゲームを難しく、かつ面白くしています。同様の操作で、助走のあとに槍を投げるステージもあります。

## アルゴリズム Algorithm

ジャンプ角度調整を実現するには、キャラクターの状態をいくつかに分けて管理する必要があります(Fig. 2-60)。ここでは状態を表すJumpStateという変数を用意しました。そして、キャラクターの状態を次の3種類に分けます。

Fig. 2-60 キャラクターの3つの状態

- 地上状態（JumpState=0）
- 角度調整状態（JumpState=1）
- ジャンプ状態（JumpState=2）

地上状態では、レバーやボタンの操作でキャラクターをX方向に移動させることができます（Fig. 2-61）。これはジャンプ前の助走に相当します。ジャンプボタンを押したら、角度調整状態に移行します。

角度調整状態では、ボタンを押している間は角度を調整することができます（Fig. 2-62）。ボタンを放すと、ジャンプ状態に移行します。

ジャンプ状態になると、キャラクターはいったん上昇したあとに、重力に引かれてだんだん落下します（Fig. 2-63）。そして、地上に到達したら最初の地上状態に戻ります。

ジャンプの飛距離をジャンプ角度によって変えるには、角度に応じてX方向とY方向の速度を計算します（Fig. 2-64）。最初に、キャラクターのX方向の移動速度とジャンプの初速度から、速度の大きさを計算します。X方向の移動速度をVX、初速度をjump_speed、速度の大きさをJumpSpeedとすると、

```
JumpSpeed=sqrt(VX*VX+jump_speed*jump_speed)
```

**Fig. 2-61** 地上状態

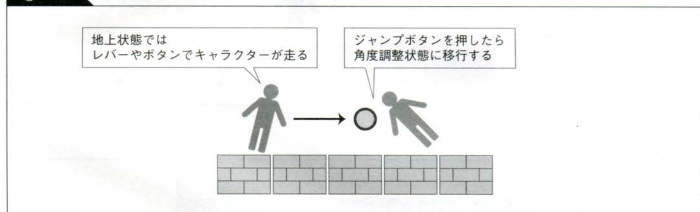
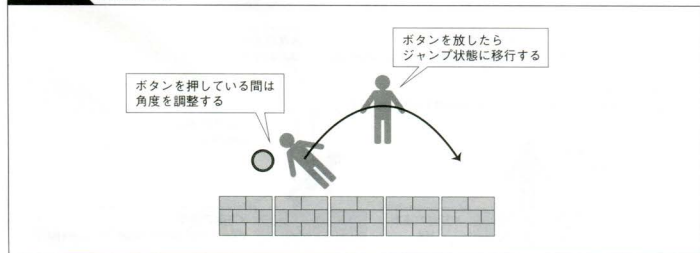



**Fig. 2-62** 角度調整状態

となります。sqrtは平方根を求める関数です。

ジャンプの角度が決まったら、その角度と速度の大きさJumpSpeedを使って、X方向とY方向の速度を計算します。角度をJumpAngle、Y方向の速度をVYとすると、

```
VX=JumpSpeed*cos(JumpAngle)
VY=-JumpSpeed*sin(JumpAngle)
```

となります。Y方向の速度は上昇する方向なので、符号をマイナスにしています。

これで、ジャンプ角度に応じて飛距離を変えることができます。この方法では、X方向の速度が大きいほど、ジャンプの高さも高くなります。これは現実世界でジャンプするときの状況とは違うのですが、速く走るほど高く跳べるようなゲームバランスにすることができます。

**Fig. 2-63** ジャンプ状態

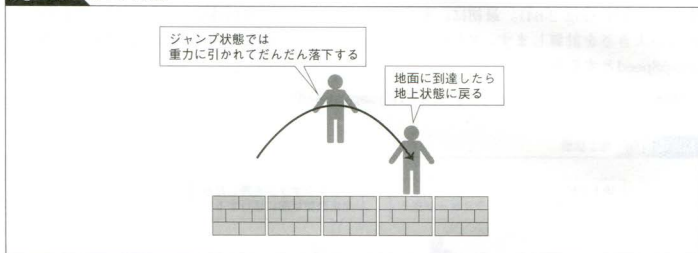
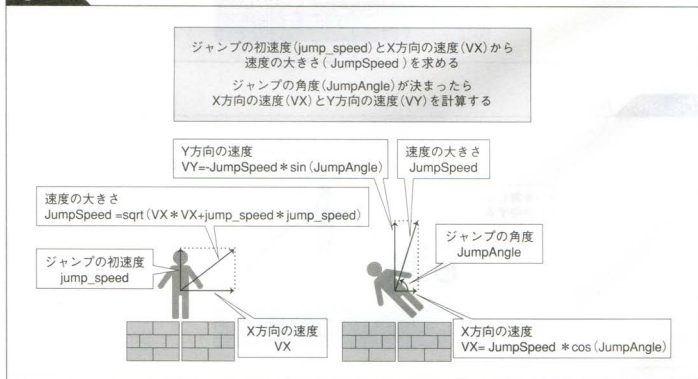



**Fig. 2-64** ジャンプ角度に応じた速度の計算

## ⊕プログラム Program

List 2-7はジャンプ角度調整のプログラムです。このプログラムでは「連打ダッシュ(→ p. 27)」の処理を組み合わせて、ボタンの連打でキャラクターを助走させることにしました。ボタンを速く連打するほどキャラクターが速く走り、ジャンプの飛距離も長くなります。

ジャンプの角度を表すJumpAngleは、0(水平)から0.25(垂直)の範囲で表すことにしました。0は0度、0.25は90度に相当します。cos関数やsin関数には角度をラジアンで与えるので、

```
VX=JumpSpeed*cosf(JumpAngle*D3DX_PI*2);
VY=-JumpSpeed*sinf(JumpAngle*D3DX_PI*2);
```

のように、D3DX_PI(πを表すDirectXの定数)と2を乗算して角度を換算しています。なお、cosfとsinfはfloat型向けのcos関数とsin関数です。

**List 2-7** ジャンプ角度調整(CJumpAngleManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // X方向の移動スピード
    float run_speed=0.6f;

    // X方向の加速度
    // 加速と減速に使う
    float run_accel=0.06f;

    // ジャンプの初速度
    float jump_speed=-0.5f;

    // ジャンプ中の加速度
    float jump_accel=0.02f;

    // ジャンプ角度の最大値
    float jump_angle_max=0.25f;

    // 1フレームごとのジャンプ角度の増分
    float jump_angle_add=0.005f;

    // 地面にキャラクターがいるときのY座標
    // 着地の判定に使う
    float ground_y=MAX_Y-2;

    // 状態に応じた処理を行う
    switch (JumpState) {

        // 地上状態
        case 0:
```

**List 2-7**

```
		// 連打ダッシュの処理
		// ボタンを押した瞬間に加速し、それ以外の場合は減速する
		if (!PrevButton && is->Button[1]) {
			VX+=run_accel;
		} else {
			VX-=run_accel*0.05f;
		}

		// X方向の速度が一定範囲内になるように補正する
		if (VX<0) VX=0;
		if (VX>run_speed) VX=run_speed;

		// 直前のボタンの入力状態を保存する
		// この情報はボタンを押した瞬間かどうかを判定するために使う
		PrevButton=is->Button[1];

		// ジャンプボタンを押したときの処理
		// X方向の速度とジャンプの初速度から、速度の大きさを計算する
		// そして、X方向の速度・ジャンプ角度・ジャンプ距離を初期化し、
		// 角度調整状態に移行する
		if (is->Button[0]) {
			JumpSpeed=sqrtf(VX*VX+jump_speed*jump_speed);
			VX=0;
			JumpAngle=0;
			JumpDist=0;
			JumpState=1;
		}
		break;

	// 角度調整状態
	case 1:

		// ボタンを放したり、角度が一定値に達した場合には、
		// ジャンプ状態へ移行する
		// 速度の大きさとジャンプの初速度から、
		// X方向とY方向の速度を計算する
		if (!is->Button[0] || JumpAngle==jump_angle_max) {
			VX=JumpSpeed*cosf(JumpAngle*D3DX_PI*2);
			VY=-JumpSpeed*sinf(JumpAngle*D3DX_PI*2);
			JumpState=2;
		} else

		// ジャンプ角度を増加させるとともに、
		// 角度が一定値を超えないように補正する
		{
			JumpAngle+=jump_angle_add;
			if (JumpAngle>jump_angle_max) JumpAngle=jump_angle_max;
		}
		break;
```

```
    // ジャンプ状態
    case 2:

        // ジャンプの飛距離を増加させる
        // このプログラムではジャンプの飛距離を数値で画面に表示する
        JumpDist+=VX;

        // Y方向の速度に加速度を加える
        VY+=jump_accel;

        // Y座標を更新する
        Y+=VY;

        // キャラクターが落下中(Y方向の速度が正の値)で、
        // かつ地面にキャラクターがいるときのY座標に達していたら、
        // 着地したと判定する
        // Y方向の速度を0にして、
        // Y座標を地面にいるときの座標に設定する
        // そして、地上状態へ移行する
        if (VY>0 && Y>=ground_y) {
            VX=0;
            Y=ground_y;
            JumpState=0;
        }
        break;
    }

    // キャラクターを傾けて表示する
    // 地上状態ではX方向の速度に応じて、
    // 角度調整状態とジャンプ状態ではジャンプの角度に応じて、
    // キャラクターを表示する角度を決める
    if (JumpState==0) Angle=VX/run_speed*0.1f; else Angle=-JumpAngle;

    return true;
}
```

## SAMPLE

「JUMP ANGLE」はジャンプの角度調整のサンプルです。ダッシュボタンの連打でキャラクターが前進し、ジャンプボタンでジャンプします。ジャンプボタンを押し続けた時間でジャンプの角度が変わります。また、キャラクターの移動速度が速いほど遠くにジャンプできます。

**JUMP ANGLE** → p. 393

## 床アタック

床の下でジャンプして、頭上の床を押し上げるアクションです。押し上げられた床の上にいる敵を弾き飛ばしたり、転倒させたりすることができます。押し上げた床の種類に応じて、アイテムを出現させる (→ p. 375) ゲームもあります。

床の下でジャンプすると、キャラクターの頭で床を押し上げることができます (Fig. 2-65)。このとき、押し上げられた床の上に敵がいると、その敵は弾き飛ばされて、転倒します (Fig. 2-66)。

床アタックを採用したゲームには、「マリオブラザーズ」があります。このゲームではジャンプで床を押し上げることで、床の上にいる敵を転倒させて行動不能にします。そのあとで敵に接近して、転倒した敵を蹴り飛ばすことで、敵を完全に倒すことができます。なお、転倒した敵の下にある床を押し上げると、転倒した敵が逆に起き上がってきます。

**Fig. 2-65** 頭で床を押し上げる

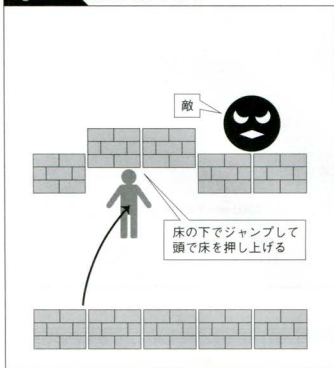
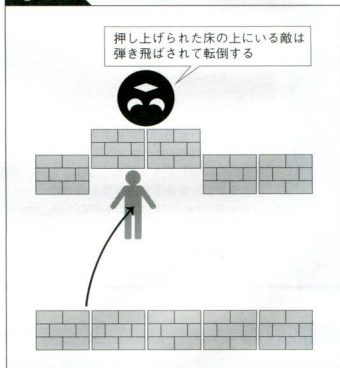



**Fig. 2-66** 敵を転倒させる

### アルゴリズム　Algorithm

床アタックを実現するには、まずはキャラクターが床を押し上げたかどうかを判定する必要があります (Fig. 2-67)。そのためには、床とキャラクターの間で当たり判定処理を行います。床を押し上げるための条件は次のとおりです。

・キャラクターと床のX座標の差分が一定範囲内である
・キャラクターと床のY座標の差分が一定範囲内である

これらの条件に加えて、次のような条件を追加してもよいでしょう。

・キャラクターがジャンプの上昇中である

この条件は必須ではありませんが、キャラクターが上昇しているときにぶつかった床だけを押し上げることになるため、動きが自然になります。

これらの条件を用いて、すべての床について、キャラクターがその床を押し上げたかどうかを判定します。床が押し上げられたら、床が押し上げられた様子を動きで表現します。これはいろいろな表現方法がありますが、例えば床をジャンプさせるような動きをするとよいでしょう (Fig. 2-68)。

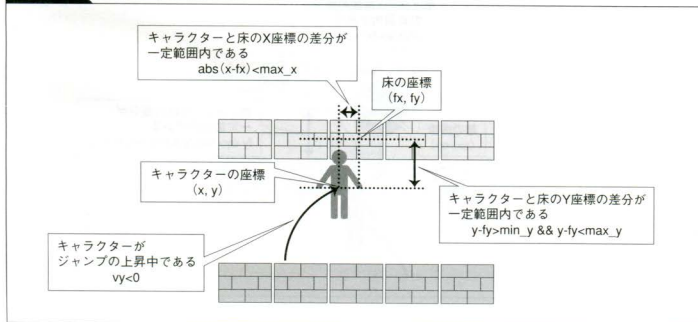
まず、床に対して上向きの初速度を与えることによって、床を上昇させます。そして、下向きの加速度を加えることによって、いったん上昇させたあとにだんだん落下させます。これはちょうど、キャラクターをジャンプさせるときの処理と同じです。

このほかに、床が押し上げられる様子をグラフィックで表現する方法もあります。例えば「マリオブラザーズ」では、押し上げられている床を表現する専用のグラフィックを用意しています。

次に、押し上げられた床によって、敵が弾き飛ばされるかどうかを判定します (Fig. 2-69)。これは敵と床の間で当たり判定処理を行います。敵を弾き飛ばすための条件は次のとおりです。

・敵と床のX座標の差分が一定範囲内である
・敵と床のY座標の差分が一定範囲内である

これらの条件に加えて、以下の条件を追加してもよいでしょう。

**Fig. 2-67** 床を押し上げるための条件

・床が上昇中である

この条件がないと、下降中の床の上をたまたま通りかかった敵も弾き飛ばされてしまいます。

これらの条件を使って、すべての敵と床の組み合わせについて、敵が床に弾き飛ばされるかどうかを判定します。弾き飛ばされた場合には、敵が飛んだり転倒したりする様子を動きで表現します。これについてもいろいろな方法がありますが、例えば敵をジャンプさせながら逆さまにするような動きをさせてもよいでしょう(Fig. 2-70)。

**Fig. 2-68** 床を押し上げる処理

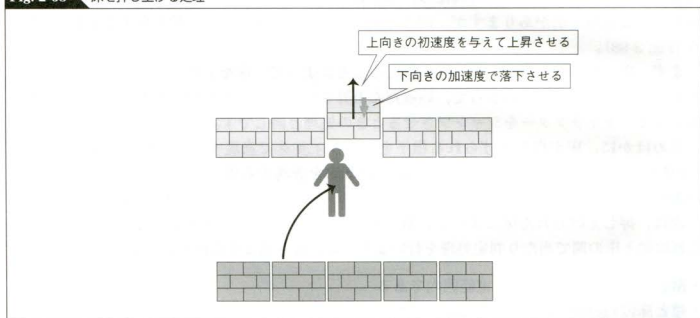



**Fig. 2-69** 敵を転倒させるための条件

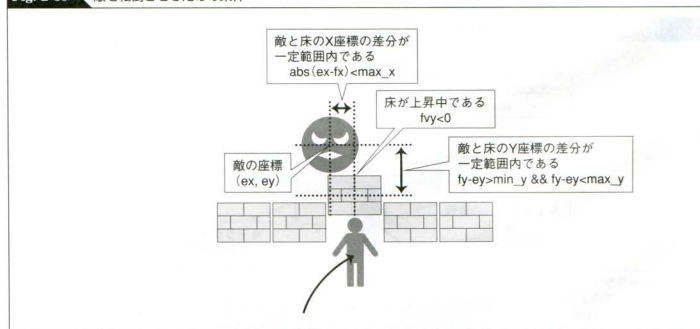
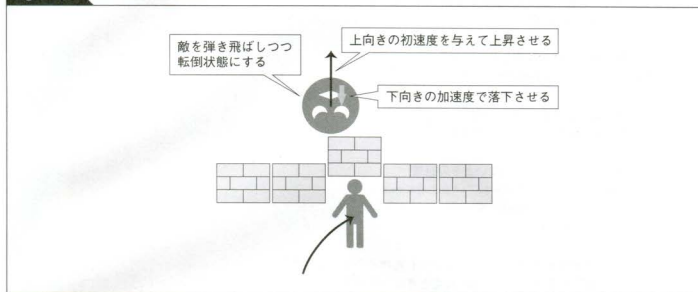



**Fig. 2-70** 敵を転倒させる処理

この動きは、床を押し上げる動きとほぼ同じです。最初に、敵に対して上向きの初速度を与えることによって、敵を上昇させます。そして、下向きの加速度を加えることによって、いったん上昇させたあとに落下させます。同時に、敵のグラフィックを逆さまに表示すれば、敵が床に弾き飛ばされて転倒したような動きになります。

## プログラム Program

List 2-8は床アタックのプログラムです。キャラクターを動かす処理は、「固定長ジャンプ(→ p. 60)」の処理とまったく同じなので掲載は省略しました。ここでは掲載したのは、床を動かす処理と、敵を動かす処理の部分です。

ここでは床を動かす処理のなかに、キャラクターに押し上げられる処理を記述しました。また、敵を動かす処理のなかに、床に弾き飛ばされる処理を記述しています。

このサンプルでは、転倒してから一定時間が経過すると、敵が自動的に復活して起き上がるようにしました。「マリオブラザーズ」でも、転倒した敵を放置しておくと、一定時間後に起き上がります。

**List 2-8** 床アタック(CFloorAttackFloorクラス、CFloorAttackEnemyクラス)

```
// 床の移動処理を行うMove関数
bool CFloorAttackFloor::Move(const CInputState* is) {

    // キャラクターと床のX座標の差分の最大値
    float max_x=0.8f;

    // キャラクターと床のY座標の差分の最小値
    float min_y=0.4f;
```

### List 2-8

```
// キャラクターと床のY座標の差分の最大値
float max_y=1.0f;

// 床が押し上げられるときの初速度
float attack_speed=-0.2f;

// 床を落下させるための加速度
float attack_accel=0.02f;

// 床が押し上げられていないときの処理
// Attackは床が押し上げられているときにtrue、
// 押し上げられていないときにfalseになるフラグ
if (!Attack) {

    // 床がキャラクターに押し上げられたかどうかを判定する
    // 以下の条件をすべて満たしたときに、床が押し上げられる
    // ・キャラクターがジャンプの上昇中またはY方向の速度が0である
    // ・キャラクターと床のX座標の差分が一定範囲内である
    // ・キャラクターと床のY座標の差分が一定範囲内である
    if (
        Man->VY<=0 &&
        abs(Man->X-X)<max_x &&
        Man->Y-Y>min_y && Man->Y-Y<max_y
    ) {
        // 床が押し上げられたときには、
        // キャラクターの上昇を中断させ、
        // 床に初速度を与えて上昇させる
        // また、床が押し上げられたことを示すフラグをtrueにする
        Man->VY=0;
        VY=attack_speed;
        Attack=true;
    }
} else

// 床が押し上げられているときの処理
{
    // Y方向の速度に加速度を加える
    VY+=attack_accel;

    // Y座標を更新する
    Y+=VY;

    // 床が最初のY座標(OriginalY)よりも下まで落下したら、
    // 床を最初のY座標に戻す
    // また、床が押し上げられたことを示すフラグをfalseにする
    if (Y>=OriginalY) {
        Y=OriginalY;
        Attack=false;
    }
}
```

```
    return true;
}

// 敵の移動処理を行うMove関数
bool CFloorAttackEnemy::Move(const CInputState* is) {

    // 敵と床のX座標の差分の最大値
    float max_x=0.8f;

    // 敵と床のY座標の差分の最小値
    float min_y=0.4f;

    // 敵と床のY座標の差分の最大値
    float max_y=1.0f;

    // 敵が弾き飛ばされるときの初速度
    float jump_speed=-0.3f;

    // 敵を落下させるための加速度
    float jump_accel=0.03f;

    // 敵が転倒している時間
    int sleep_time=60;

    // 弾き飛ばされていないときの処理
    if (!Jump) {

        // 敵が転倒していないときには、X方向に移動させる
        // 画面の右端からはみ出したときには、左端に戻す
        if (!Sleep) {
            X+=VX;
            if (X>MAX_X) X=-1;
        } else

        // 敵が転倒しているときには、
        // 一定時間が経過するまで待機させる
        {
            Time++;
            if (Time>=sleep_time) Sleep=false;
        }

        // すべての床との間で当たり判定処理を行い、
        // 床に弾き飛ばされたかどうかを判定する
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (mover->Type==1) {
                CFloorAttackFloor* floor=(CFloorAttackFloor*)mover;

                // 敵が床に弾き飛ばされたかどうかを判定する。
                // 以下の条件をすべて満たしたときに、敵が弾き飛ばされる
```

## List 2-8

```
				// ・敵と床のX座標の差分が一定範囲内である
				// ・敵と床のY座標の差分が一定範囲内である
				if (
					abs(floor->X-X)<max_x &&
					floor->Y-Y>min_y && floor->Y-Y<max_y
				) {
					// 敵が弾き飛ばされたときには、
					// 敵に初速度を与えて上昇させる
					// また、転倒していないときは転倒させ、
					// 転倒しているときは復活させる
					// タイマーを設定し、
					// 弾き飛ばされているかどうかのフラグも設定する
					VY=jump_speed;
					Sleep=!Sleep;
					Time=0;
					Jump=true;
				}
			}
		}
} else

// 弾き飛ばされているときの処理
{
	// 移動してきた敵が転倒したときには、
	// 移動の勢いで横に弾き飛ばされるようにする
	// X座標を更新し、画面からはみ出さないように補正する
	if (Sleep) {
		X+=VX;
		if (X>MAX_X) X=-1;
	}

	// Y方向の速度に加速度を加える
	VY+=jump_accel;

	// Y座標を更新する
	Y+=VY;

	// 敵が最初のY座標（OriginalY）よりも下まで落下したら、
	// 敵を最初のY座標に戻す
	// また、敵が弾き飛ばされたことを示すフラグをfalseにする
	if (Y>=OriginalY) {
		Y=OriginalY;
		Jump=false;
	}
}

// 敵が転倒しているときには逆さまに、
// 転倒していないときには通常の向きで、
// グラフィックを表示する
Angle=Sleep?0.5f:0;
```

```
    return true;
  }
```

SAMPLE

「FLOOR ATTACK」は床アタックのサンプルです。左右のレバーでキャラクターを動かし、ボタンでジャンプします。ジャンプによって、上のフロアの床を押し上げることができます。うまく敵のいる床を押し上げれば、敵を弾き飛ばすことができます。

**FLOOR ATTACK** → p. 393

# 踏みつけ

敵の上にジャンプで飛び降りて、敵を踏みつけるアクションです。敵の真上に飛び降りれば踏みつぶすことができますが、飛び降りる位置が悪いと逆にダメージを受けてしまいます。速く動く敵を狙って踏みつけるのは難しく、スリリングなアクションになります。

敵を踏みつけるには、まず敵を狙ってジャンプします (Fig. 2-71)。うまく敵の上に着地すると、踏みつけることができます (Fig. 2-72)。

踏みつけを採用したゲームには、「スーパーマリオブラザーズ」などがあります。このゲームでは敵を踏みつけると、敵をつぶして倒すことができます。また、踏みつけた敵を足場がわりにして、通常のジャンプでは届かない場所にいくアクションもあります。

**Fig. 2-71** 敵を狙ってジャンプする

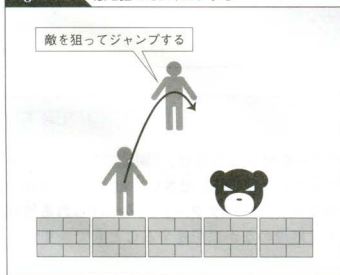



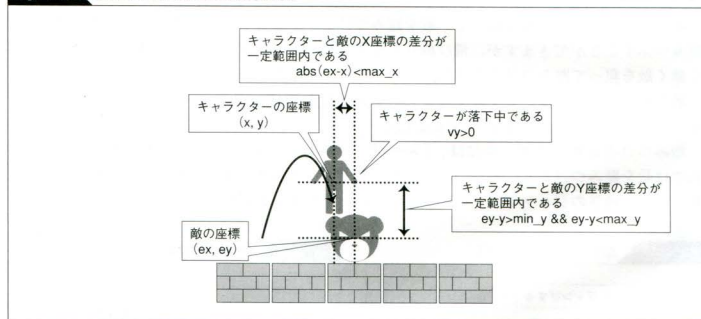
**Fig. 2-72** 敵を踏みつける

## アルゴリズム Algorithm

踏みつけを実現する際のポイントは、キャラクターが敵を踏みつけたかどうかの判定処理です（Fig. 2-73）。敵を踏みつけるには、次のような条件を満たす必要があります。

・キャラクターと敵のX座標の差分が一定範囲内である
・キャラクターと敵のY座標の差分が一定範囲内である
・キャラクターが落下中である

キャラクターとすべての敵の間で、これらの条件が成立するかどうかを調べます。条件が成立したら、その敵は踏みつけられたということです。踏みつけられたあとの動きはいろいろと考えられますが、例えば敵がつぶれるグラフィックを表示したあとに、敵を消滅させればよいでしょう。

Fig. 2-73 敵を踏みつけたかどうかの判定処理

## プログラム Program

List 2-9は踏みつけのプログラムです。キャラクターを動かす処理は、「固定長ジャンプ（→p. 60）」の処理とまったく同じなので、そちらのプログラムをご覧ください。List 2-9は踏みつけられる敵の処理部分です。ここでは、敵の処理のなかにキャラクターに踏みつけられる処理を記述しました。この処理は、キャラクターを動かす処理のなかに記述することもできます。

**List 2-9** 踏みつけ(CStompEnemyクラス)

```
// 敵の移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 敵と床のX座標の差分の最大値
    float max_x=0.8f;

    // 敵と床のY座標の差分の最小値
    float min_y=0.1f;

    // 敵と床のY座標の差分の最大値
    float max_y=1.0f;

    // 敵がつぶれるスピード
    float smash_speed=-0.1f;

    // 敵が踏みつけられる前の処理
    // Stompedは踏みつけられたかどうかを表すフラグ
    if (!Stomped) {

        // X座標の更新
        X+=VX;

        // 画面の右端からはみ出したときには、左端に戻す
        if (X>MAX_X) X=-1;

        // 踏みつけられたかどうかの判定処理
        // 次の条件をすべて満たしたときに、踏みつけられたと判定する
        // ・キャラクターが落下中である
        // ・キャラクターと敵のX座標の差分が一定範囲内である
        // ・キャラクターと敵のY座標の差分が一定範囲内である
        if (
            Man->VY>0 &&
            abs(X-Man->X)<max_x &&
            Y-Man->Y>min_y && Y-Man->Y<max_y
        ) {
            // 踏みつけられたことを示すフラグをtrueにする
            Stomped=true;
        }
    } else

    // 敵が踏みつけられた後の処理
    {
        // 敵がつぶれる動きを表現する
        // 敵の上下方向のサイズを小さくするとともに、
        // 敵の表示位置を下に移動させる
        // Hは敵の上下方向のサイズ、YはY座標を表す
        H+=smash_speed;
        Y-=smash_speed*0.5f;
```

List 2-9

```
        // 敵が完全につぶれたときの処理
        // 敵の上下方向のサイズが0未満になったら、
        // 敵のサイズや座標を初期化して、
        // 再び画面の左端から出現させる
        if (H<0) {
            Stomped=false;
            H=1;
            Y=MAX_Y-2;
            X=-1;
        }
    }

    return true;
}
```

SAMPLE

「STOMP」は踏みつけのサンプルです。左右のレバーでキャラクターを動かし、ボタンでジャンプします。敵キャラクターの上に着地すると、踏みつけることができます。

**STOMP** → p. 394

## 踏み切り板

踏み切り板を狙ってジャンプし、踏み切り板を踏んだ瞬間にタイミングよくボタンを押すと、普通よりも高く跳べるというアクションです。雰囲気は跳び箱の踏み切り板に似ています。深い谷や大きな池の前に踏み切り板が設置されていることが多く、踏み切り板をうまく使わないと、障害物を跳び越えることができません。

踏み切り板を使うには、まず板の手前から踏み切り板を狙ってジャンプします (Fig. 2-74)。板の真上に落下するようにジャンプするのがポイントです。

キャラクターが踏み切り板の上に落下したら、タイミングよくボタンを押します (Fig. 2-75)。ジャンプに成功すると、踏み切り板を利用して普通よりもずっと高く跳ぶことができます (Fig. 2-76)。

踏み切り板を採用したゲームには、「パックランド」や「メトロクロス」があります。「パックランド」の場合には、大きな池などの前に踏み切り板が設置されていて、タイミングよく跳ばないと、飛距離が足りなくて池に落ちてしまいます。「メトロクロス」では、踏み切り板を使って障害物を跳び越えられるほか、ゴールのゲートを跳び越えるとボーナスが入るようにもなっています。

Fig. 2-74 踏み切り板を狙ってジャンプする

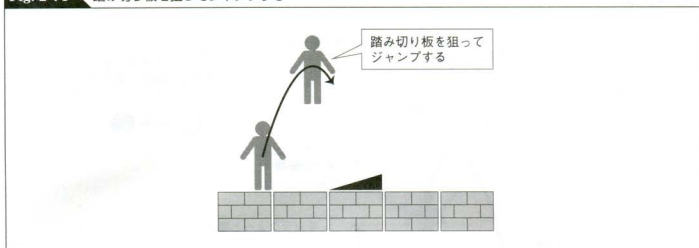



Fig. 2-75 踏み切り板の上でボタンを押す

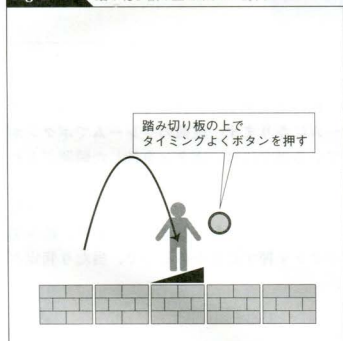
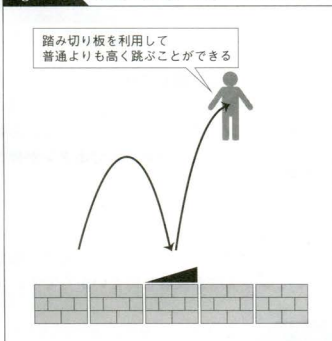
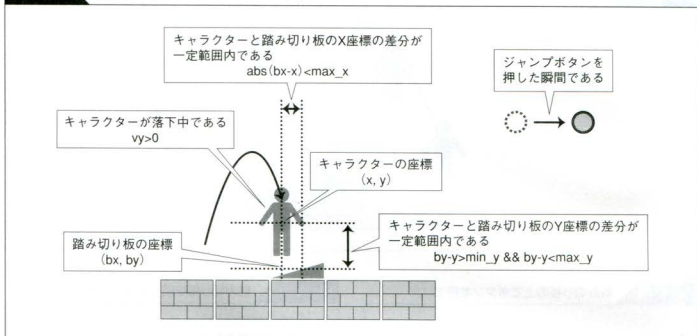



Fig. 2-76 踏み切り板を使って高くジャンプする

## アルゴリズム Algorithm

踏み切り板を実現する際のポイントは、踏み切り板を使ってうまくジャンプできたかどうかの判定処理です (Fig. 2-77)。踏み切り板を使ってジャンプするためには、次のような条件を満たす必要があります。

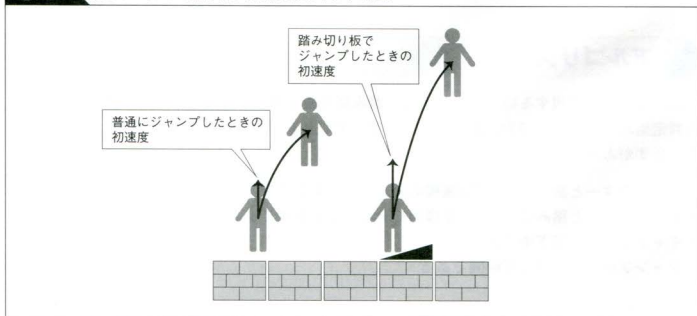
- キャラクターと踏み切り板のX座標の差分が一定範囲内である
- キャラクターと踏み切り板のY座標の差分が一定範囲内である
- キャラクターが落下中である
- ジャンプボタンを押した瞬間である

**Fig. 2-77** 踏み切り板でジャンプするための条件

ジャンプボタンに関しては、単にボタンを押しているというだけではなく、ボタンを押した瞬間かどうかを判定するとよいでしょう。踏み切り板の上でタイミングよくボタンを押したときだけジャンプを成功させると、スリルのあるゲームになります。直前のフレームでボタンが放されていて、現在のフレームでボタンが押されている場合には、ボタンを押した瞬間だとわかります。

また、キャラクターと踏み切り板のX座標とY座標の差分を比較するのは、一種の当たり判定処理です。踏み切り板の当たり判定は、ある程度大きめにしておくとよいでしょう。踏み切り板を踏むだけではなく、ちょうど踏んだ瞬間にボタンを押す必要があるので、当たり判定が小さいとジャンプが成功しにくいからです。

**Fig. 2-78** 普通のジャンプと踏み切り板を使ったジャンプの違い

踏み切り板で
ジャンプしたときの
初速度

普通にジャンプしたときの
初速度

踏み切り板でジャンプしたときには、普通のジャンプよりも大きな初速度を上方向に与えます (Fig. 2-78)。初速度が大きいほど、キャラクターは高く跳ぶことができます。

## プログラム Program

List 2-10は踏み切り板のプログラムです。ジャンプをしている処理のなかに、踏み切り板でジャンプしたかどうかの判定処理を記述しました。踏み切り板の当たり判定をいろいろと調整して、気持ちよく跳べるパラメータを探してみてください。

**List 2-10** 踏み切り板(CStampBoardManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.2f;

    // ジャンプの初速度
    float jump_speed=-0.3f;

    // ジャンプの加速度
    float jump_accel=0.02f;

    // 地面にキャラクターがいるときのY座標
    // 着地の判定に使う
    float ground_y=MAX_Y-2;

    // 踏み切り板でジャンプしたときの初速度
    float board_jump_speed=-0.6f;

    // キャラクターと踏み切り板のX座標の差分の最大値
    float max_x=1.0f;

    // キャラクターと踏み切り板のY座標の差分の最小値
    float min_y=-1.0f;

    // キャラクターと踏み切り板のY座標の差分の最大値
    float max_y=0.8f;

    // ジャンプしていないときの処理
    if (!Jump) {

        // レバーを右に入れているときには、
        // 右方向の速度を設定する
        VX=0;
        if (is->Right) VX=speed;
```

List 2-10

```
        // ボタンを押したときにはジャンプする
        // 普通のジャンプの初速度を設定する
        if (is->Button[0]) {
            Jump=true;
            VY=jump_speed;
        }
    } else

    // ジャンプしているときの処理
    {
        // Y方向の速度に加速度を加える
        VY+=jump_accel;

        // Y座標の更新
        Y+=VY;

        // 踏み切り板でジャンプしたかどうかの判定処理
        // 次の条件を満たしたときには踏み切り板を使ってジャンプする
        // ・キャラクターが落下中である
        // ・ジャンプボタンを押した瞬間である
        // ・キャラクターと踏み切り板のX座標の差分が一定範囲内である
        // ・キャラクターと踏み切り板のY座標の差分が一定範囲内である
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (
                mover->Type==1 &&
                VY>0 &&
                !PrevButton && is->Button[0] &&
                abs(mover->X-X)<max_x &&
                mover->Y-Y>min_y && mover->Y-Y<max_y
            ) {
                // ジャンプに成功したら、
                // 普通よりも大きなジャンプの初速度を設定する
                VY=board_jump_speed;
            }
        }

        // キャラクターが落下中(Y方向の速度が正の値)で、
        // かつ地面にキャラクターがいるときのY座標に達していたら、
        // 着地したと判定する
        // ジャンプ状態から通常状態に戻って、
        // Y座標を地面にいるときの座標に設定する
        if (VY>0 && Y>=ground_y) {
            Jump=false;
            Y=ground_y;
        }
    }

    // 直前のボタンの状態を保存する
    // ボタンを押した瞬間を判定するために使う
```

```
    PrevButton=is->Button[0];

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「STAMP BOARD」は踏み切り板のサンプルです。右のレバーでキャラクターが前進し、ボタンでジャンプします。踏み切り板の上に着地するようにタイミングを調整してジャンプし、踏み切り板を踏んだ状態で再度ジャンプボタンを押せば、普通より高く飛び上がることができます。

**STAMP BOARD → p. 394**

## まとめ Stage02

本章ではアクションゲームに欠かせない「ジャンプ」について解説しました。ジャンプの動きを制御するには速度や加速度を使い、着地や飛び降りを扱うには当たり判定処理が必要です。キャラクターをちょっとジャンプさせるだけでも、考えなければならないことは非常に多くあります。しかし、ジャンプの処理が理解できれば、アクションゲームの最も重要な処理は理解できたといってもよいでしょう。

というわけで、「ジャンプを使うゲームを作るならば、まずジャンプから作ろう！」というのが本章のまとめです。

アクションゲームのステージには、さまざまな「仕掛け」があります。キャラクターは数多くの仕掛けが組み込まれたステージのなかを、ダッシュやジャンプを駆使して切り抜けていきます。魅力的な仕掛けは、ゲームを面白くするだけではなく、そのゲームの個性を強烈にアピールします。

# 仕掛け

## Gimmick

ActionGame Algorithm Maniax

# Stage 03

# ロープ

キャラクターがつかまって移動することができる仕掛けです。ロープにつかまって左右に移動したり、落下する途中でロープをつかんで止まったりといったアクションが可能です。

ロープには、横方向と縦方向がありますが、ここでは横方向のロープについて解説します。縦方向のロープは、横方向のロープや「はしご(→ p. 128)」の応用で作ることができます。

キャラクターはロープにつかまることができます(Fig. 3-1)。ロープにつかまっているときには、レバーを左右に入力して、キャラクターを左右に移動させることが可能です(Fig. 3-2)。

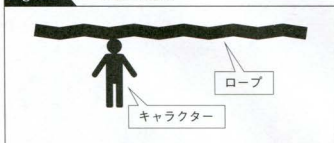
レバーを下に入れると、キャラクターはロープを放して落下します(Fig. 3-3)。一度落下を始めたら、あとはレバーを下に入れていなくても落下を続けます。

落下中にレバーを下に入れておらず、かつロープがキャラクターのすぐ近くにあれば、再びロープにつかまることができます(Fig. 3-4)。ロープが張り巡らされたステージでは、ロープにつかまって左右に移動し、ロープを放して落下することによって、ステージ内を自由に動き回ることができます。

ロープを採用したゲームの例としては、「ロードランナー」があります。このゲームのロープは横方向です。ロープを使って左右に移動したり、ロープにつかまって落下を止めたりすることができます。ステージによってはロープが迷路のように張り巡らされており、非常に複雑な

**Fig. 3-1** ロープにつかまる

ロープ
キャラクター

**Fig. 3-2** ロープを使って左右に移動する

ロープにつかまって
左右に移動することが
できる

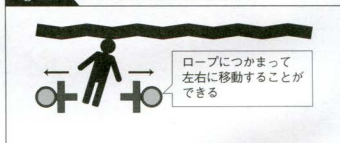



**Fig. 3-3** ロープを放して落下する

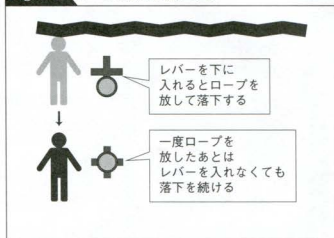
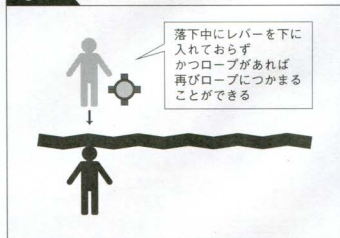



**Fig. 3-4** 落下中にロープにつかまる

ロープの網を利用して、金塊を集めたり、敵から逃げたりといったアクションが楽しめます。

また「ドンキーコングJr.」もロープを採用しています。こちらは縦方向のロープで、性質としては「はしご（→ p. 128）」に近い仕掛けです。このゲームで面白いのは、1本のロープにつかまったときと、同時に2本のロープにつかまったときでは、キャラクターがロープを上下するスピードが変わることです。1本のロープにつかまったときの方が、キャラクターの当たり判定が小さくなるので敵にやられにくいのですが、上下のスピードは遅くなります。そこで、敵がいないときを見計らって2本のロープにつかまり、敵が近づいたら1本のロープを放すという、独特のアクションを行います。

## アルゴリズム Algorithm

ロープを実現するときのポイントは、キャラクターがロープにつかまっているかどうかの判定処理です。これは一種の当たり判定処理です。当たり判定処理については、「ジャンプ」の章（→ p. 59）でも詳しく解説しているので、そちらもご覧ください。

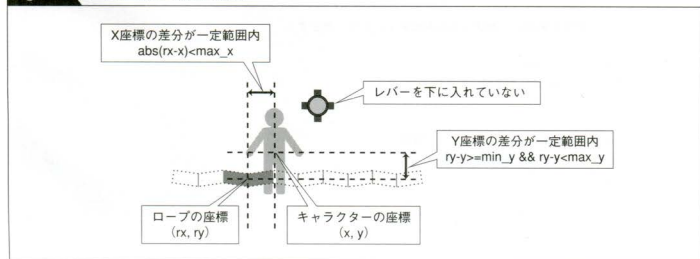
ロープにつかまるためには、次のような条件を満たす必要があります（Fig. 3-5）。

- キャラクターとロープのX座標の差分が一定範囲内
- キャラクターとロープのY座標の差分が一定範囲内
- レバーを下に入れていない

これらの条件をすべて満たしたときには、キャラクターはロープにつかまっているため落下しません。また、ロープにつかまって左右に移動することができます。

条件が満たされていないときには、キャラクターは落下します。落下中もすべてのロープに対して上記の判定を行い、ロープにつかまったときには落下を中断します。

X座標やY座標の差分の範囲は、少し広めにとっておいた方がよいでしょう。広めにしておくと、ロープにつかまりやすくなります。特にY座標の差分については、落下するキャラクターが当たり判定の範囲を通り抜けてしまわないように、キャラクターが1フレームに移動する距離よりも広くとっておく必要があります（→ p. 86）。

**Fig. 3-5** ロープにつかまるための条件

また、キャラクターがロープにつかまったときには、キャラクターのY座標をロープのY座標に合わせます。つかまった瞬間のキャラクターのY座標は場合によって異なるので、ロープにぴったりとつかまった様子を表すためには、Y座標を合わせる必要があります。

## プログラム Program

List 3-1はロープのプログラムです。画面内に張り巡らされたロープを使って、左右への移動と落下が可能です。キャラクターが画面の下端まで落下したときには、画面の上端から再び出現するようにしています。

**List 3-1** ロープ(CRopeManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.2f;

    // キャラクターとロープのX座標の差分の最大値
    float max_x=0.8f;

    // キャラクターとロープのY座標の差分の最小値
    float min_y=0.0f;

    // キャラクターとロープのY座標の差分の最大値
    float max_y=0.2f;

    // ロープにつかまっているときの処理
    if (VY==0) {

        // レバーの入力に応じて左右の速度を設定する
        VX=0;
        if (is->Left) VX=-speed;
        if (is->Right) VX=speed;

        // X座標を更新し、画面からはみ出さないように補正する
        X+=VX;
        if (X<0) X=0;
        if (X>MAX_X-1) X=MAX_X-1;
    }

    // ロープにつかまっているかどうかを判定し、
    // つかまっていないときには落下する
    VY=speed;
    if (!is->Down) {
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
```

```
            // レバーを下に入れておらず、
            // かつキャラクターとロープのX座標とY座標の差分が
            // 一定範囲内ならば、ロープにつかまる
            // キャラクターのY方向の速度を0にして、
            // Y座標をロープのY座標に合わせる
            if (
                mover->Type==1 &&
                abs(mover->X-X)<max_x &&
                mover->Y-Y>=min_y && mover->Y-Y<max_y
            ) {
                VY=0;
                Y=mover->Y;
                break;
            }
        }
    }

    // Y座標の更新
    Y+=VY;

    // キャラクターが画面下端から出たときには、画面上端から再出現する
    if (Y>MAX_Y) Y=-1;

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

## SAMPLE

「ROPE」はロープのアクションのサンプルです。ロープにつかまっている間は、左右のレバーでキャラクターを移動させることができます。レバーを下に入れるかロープの端を越えた場合、キャラクターは落下します。落下先にロープがあれば、キャラクターはそのロープにつかまります。レバーを下に入れっぱなしにしていると、キャラクターはいつまでも落下を続けます。

**ROPE → p. 394**

# はしご

キャラクターがつかまって移動することができる仕掛けです。はしごにつかまっている間は、自由に上下に移動することができます。また、少し左右に移動したり、はしごの上に乗ったりすることも可能です。

キャラクターははしごにつかまることができます（Fig. 3-6）。はしごにつかまっているときには、レバーを上下に入れることによって、キャラクターを上下に移動させることが可能です（Fig. 3-7）。

また、はしごの上に乗ったり（Fig. 3-8）、はしごを使って左右に移動することもできます

**Fig. 3-6** はしごにつかまる

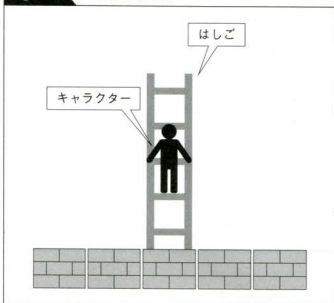



**Fig. 3-7** はしごを使って上下に移動する

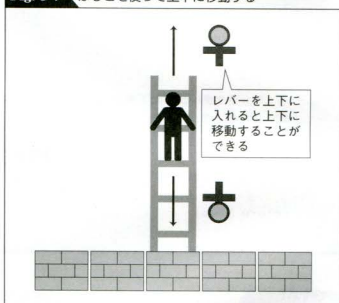



**Fig. 3-8** はしごの上に乗る

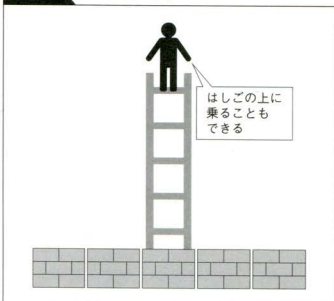



**Fig. 3-9** はしごを使って左右に移動する

Fig. 3-10 はしごから落ちる

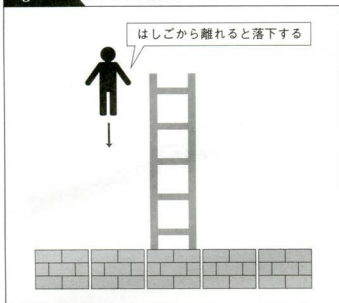



Fig. 3-11 はしごにつかまって落下を止める

(Fig. 3-9)。多くの場合、はしごは上下方向に伸びているので、左右に移動できる幅はそれほど広くはありません。はしごが隣接して配置してある場合、左右のはしごに乗り移れるゲームもあります。

はしごから離れると、キャラクターは落下します(Fig. 3-10)。再びはしごにつかまったり、はしごに乗ったりすると落下は止まります(Fig. 3-11)。

はしごを採用したゲームは非常に数多くあります。例えば「ドンキーコング」では、階層構造になったステージを、はしごを使って登っていきます。また、はしごの途中につかまって、転がってくるタルをやりすごすアクションも取り入れられています。

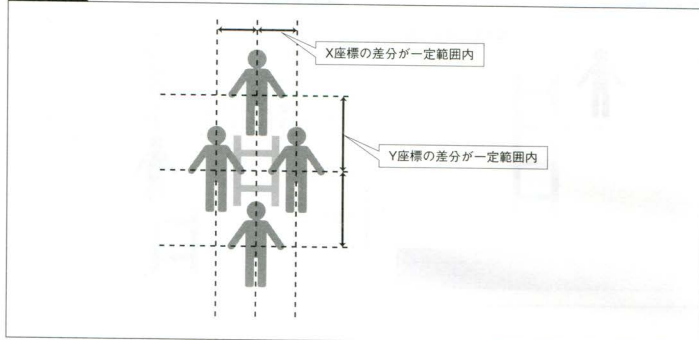
「ロードランナー」では、はしごと「ロープ(→ p. 124)」の両方が採用されています。はしごとロープはどちらもキャラクターがつかまれるという点では似ていますが、それぞれ異なる働きをします。ロープは左右にしか移動できません。逆にはしごは上下左右に移動できますが、左右に並んで配置されていることが少なくなっています。そこで、左右方向の移動にはロープを使い、上下方向の移動にははしごを使う、といった使い分けが要求されます。

## アルゴリズム Algorithm

はしごを実現する際のポイントは、はしごにつかまっているかどうかの判定処理です(Fig. 3-12)。これはロープの場合と同様に、一種の当たり判定処理になります。はしごにつかまるための条件は次のとおりです。

- X座標の差分が一定範囲内
- Y座標の差分が一定範囲内

詳しい条件式については、サンプルプログラムを参照してください。また、「ロープ(→ p. 124)」の処理も参考にしていただけるでしょう。

**Fig. 3-12** はしごにつかまるための条件

## プログラム Program

List 3-2は、はしごのプログラムです。このプログラムでは、キャラクターがはしごにつかまっているのか、床の上にいるのか、あるいは落下しているのかを判定して、状況に応じた移動処理を行います。はしごにつかまっているときには、キャラクターは上下左右に移動できます。床にいるときには、左右にのみ移動が可能です。落下しているときには、落下が終わるまでは何も操作ができません。

床に乗っているかどうかの判定は、キャラクターのY座標を使って行います。一方、床との当たり判定処理を行う方法もあります。床が何層にも重なっているステージでは、当たり判定処理を使うとよいでしょう。階層のある床の処理の詳細は「切り替わる通路 (→ p. 183)」のサンプルをご覧ください。

**List 3-2** はしご(CLadderManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.2f;

    // 地面にキャラクターがいるときのY座標
    // 着地の判定に使う
    float ground_y=MAX_Y-2;

    // キャラクターとはしごのX座標の差分の最大値
```

```
float max_x=0.6f;

// キャラクターとはしごのY座標の差分の最小値
float min_y=-1.0f;

// キャラクターとはしごのY座標の差分の最大値
float max_y=1.0f;

// はしごにつかまっているかどうか
bool ladder=false;

// 床に乗っているかどうか
bool floor=false;

// はしごにつかまっているかどうかの判定処理
// はしごとキャラクターの当たり判定処理を行う
// つかまっているときにはladderをtrueにする
for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
    CMover* mover=(CMover*)i.Next();

    if (
        mover->Type==1 &&
        abs(mover->X-X)<max_x &&
        mover->Y-Y>=min_y && mover->Y-Y<max_y
    ) {
        // 単にladderをtrueにすると、
        // はしごの上端でキャラクターが振動してしまうので、
        // はしごの上端付近にいるときには床に乗っている扱いにしている
        if (mover->Y-Y<max_y-speed) ladder=true; else floor=true;
        break;
    }
}

// 床に乗っているかどうかの判定処理
// キャラクターのY座標を調べる
// 床とキャラクターの当たり判定処理を行う方法もある
// 「切り替わる通路」を参照
if (Y>=ground_y) {
    Y=ground_y;
    floor=true;
}

// はしごにつかまっているか、床に乗っているときの処理
if (ladder || floor) {

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;
```

**List 3-2**

```
		// X座標を更新し、画面からはみ出さないように補正する
		X+=VX;
		if (X<0) X=0;
		if (X>MAX_X-1) X=MAX_X-1;

		// レバーの入力に応じて上下に移動する
		VY=0;
		if (ladder && is->up) VY=-speed;
		if (is->Down) VY=speed;

		// Y座標を更新し、
		// キャラクターが画面からはみ出さないように補正する
		Y+=VY;
		if (Y<0) Y=0;
		if (Y>MAX_Y-1) Y=MAX_Y-1;
	}

	// はしごにつかまっておらず、床にも乗っていないとき、
	// すなわち落下しているときの処理
	// レバーによる操作は不可能で、ただ落下する
	if (!ladder && !floor) {
		Y+=speed;
	}

	// X方向の速度に応じて、
	// キャラクターを傾けて表示する
	Angle=VX/speed*0.1f;

	return true;
}
```

SAMPLE

「LADDER」は、はしごのサンプルです。上下のレバーではしごを昇降することができます。つかまっている間は、はしごを左右に移動することもできます。はしごの上から飛び降りることもできます。

**LADDER** → p. 394

# トランポリン

キャラクターが上に乗るとジャンプする仕掛けです。複数の階があるステージでは、トランポリンを利用して上下の階に移動することができます。

トランポリンは床と床の間などに設置されています (Fig. 3-13)。キャラクターがトランポリンの上に乗ると、トランポリンが動き出します (Fig. 3-14)。

トランポリンの力を借りて、キャラクターは上昇します (Fig. 3-15)。キャラクターはどんどん上昇していきますが、天井にぶつかったり、画面の上端に達したりすると落下に転じます (Fig. 3-16)。

**Fig. 3-13** トランポリン

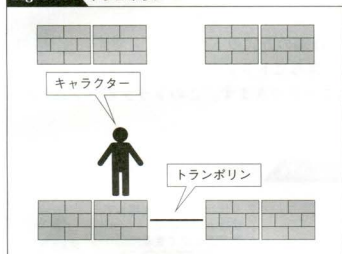



**Fig. 3-14** トランポリンに乗る

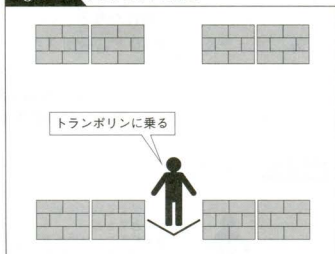



**Fig. 3-15** トランポリンを使って上昇する

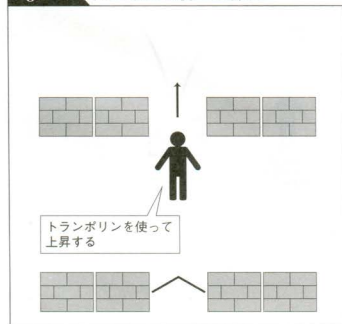
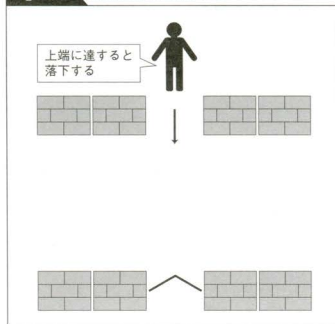



**Fig. 3-16** 上端に達すると落下する

落下したキャラクターがトランポリンに戻ると、再びトランポリンでジャンプして上昇します。トランポリンから降りないかぎり、キャラクターは上昇と落下を繰り返します。

キャラクターが床に乗れる位置で、タイミングよくレバーを左右に入力すると、床に着地することができます(Fig. 3-17)。床に着地したキャラクターは歩行状態に戻るので、レバーを使って左右に移動することができます。

2階以上にいるキャラクターがトランポリンの縦穴に落ちると、キャラクターはトランポリンの位置まで落下します。トランポリンに達すると、ジャンプして上昇します。あとはトランポリンから降りないかぎり、やはり上昇と落下の繰り返しです。

この方式のトランポリンを採用しているゲームには「マッピー」があります。「マッピー」ではトランポリンを利用して、複数の階で構成されたステージを上下に移動します。また、トランポリンで飛んでいる間は敵をすり抜けることができるため、トランポリンを使って敵をかわすことができます。ただし、「マッピー」のトランポリンは、連続して使用すると破れてしまいます。また、トランポリンに乗るときと降りるときにキャラクターが軽くジャンプしたり、トランポリンがリアルに揺れたりと、いろいろと見どころがあります。

一方、「ナッツ&ミルク」などに採用されているのは別の方式のトランポリンです(Fig. 3-18)。キャラクターがジャンプしてトランポリンに乗り、さらにトランポリンの上でタイミングよくジャンプすると、普通のジャンプよりも高く跳ぶことができます。このトランポリンは「踏み切り板(→ p. 121)」の応用で作ることができます。

**Fig. 3-17** 床に着地する

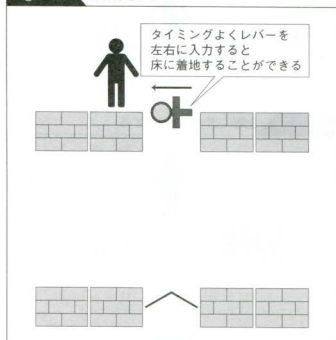
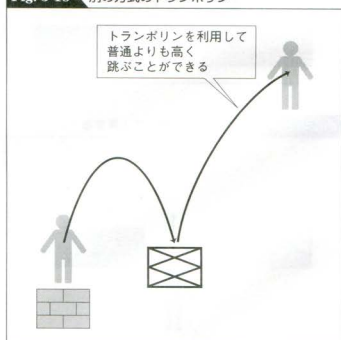



**Fig. 3-18** 別の方式のトランポリン

## アルゴリズム Algorithm

トランポリンを実現する際のポイントは、歩行している状態とトランポリンに乗っている状態を区別することです。まず、歩行しているときにトランポリンの領域に入ったら、トランポリン状態に移行します(Fig. 3-19)。トランポリンの領域は、トランポリンのすぐ上から、天井あるいは画面端まで広がっています。

トランポリン状態では、キャラクターは上昇と落下を繰り返します。そして、着地可能な領域でレバーを左右に入力したら、歩行状態に移行します(Fig. 3-20)。着地可能な領域は、各階の床の上に左右方向に広がっています。

**Fig. 3-19** トランポリンに乗る処理

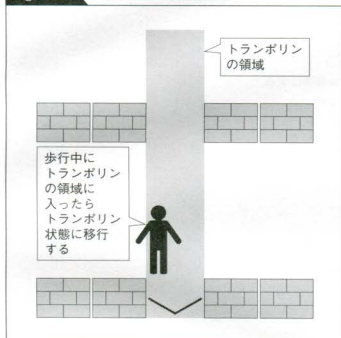
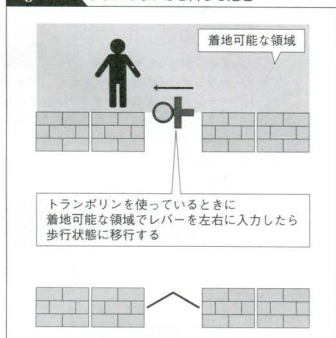



**Fig. 3-20** トランポリンから降りる処理

## プログラム Program

List 3-3はトランポリンのプログラムです。このプログラムでは「マッピー」のルールに合わせて、トランポリンで上昇しているときだけ、左右にレバーを入れると床に着地できるようにしました。

**List 3-3** トランポリン(CTrampolineManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.2f;
```

**List 3-3**

```
// トランポリンの位置を計算するための定数
int trampoline_x=5;

// 床の位置を計算するための定数
int floor_y=3;

// 歩行状態の処理
if (!Trampoline) {

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標の更新
    X+=VX;

    // トランポリンに乗る処理
    // トランポリンの領域に入ったら、トランポリン状態にする
    // トランポリンの領域かどうかは座標で判定するが、
    // ここではトランポリンを規則的に配置しているため、
    // forループを使ってトランポリンの座標を計算している
    // トランポリンの配置が複雑なときには、
    // 座標をデータ化しておくとよい
    for (int i=0; i<4; i++) {
        float tx=i*5;
        if (abs(X-tx)<0.1f) {

            // トランポリンに乗ったら、座標や速度を設定する
            X=tx;
            VX=0;
            VY=speed;
            Trampoline=true;
            break;

        }
    }
} else

// トランポリン状態の処理
{
    // Y座標の更新
    Y+=VY;

    // 画面の上端または下端に達したら、
    // 速度の方向を逆にする
    if (Y<=0) VY=speed;
    if (Y>=MAX_Y-2) VY=-speed;

    // トランポリンから降りる処理
    // 着地の領域に入ったら、歩行状態にする
```

```
        // 着地の領域かどうかは座標で判定するが、
        // ここでは床を規則的に配置しているため、
        // forループを使って床の座標を計算している
        // 床の配置が複雑なときには、
        // 座標をデータ化しておくとよい
        for (int i=0; i<4; i++) {
            float fy=MAX_Y-2-i*3;

            // トランポリンから降りるためには以下の条件が必要
            // ・上昇中である
            // ・着地の領域に入っている
            // ・画面の左端以外でレバーを左に入れているか、
            //   画面の右端以外でレバーを右に入れている
            if (
                VY<0 &&
                abs(Y-fy)<0.1f &&
                (X>0 && is->Left || X<MAX_X-1 && is->Right)
            ) {

                // トランポリンから降りたら、座標や速度を設定する
                Y=fy;
                VY=0;
                Trampoline=false;
                break;
            }
        }
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「TRAMPOLINE」はトランポリンのサンプルです。左右のレバーでキャラクターを移動させることができます。キャラクターがトランポリンの上に乗ると上昇を始め、画面最上部に達すると落下します。上昇中は、タイミングよくレバーを左右に入れることで、左右の床に乗ることができます。落下中は、レバー操作を受け付けません。

**TRAMPOLINE** → p. 394

## ⊕抜ける床

キャラクターが上を通過すると床が抜ける仕掛けです。多くの場合は普通の床と抜ける床の見分けがつくようになっていますが、わざとまぎらわしくしてある場合もあります。抜ける床があるステージでは、床をよく見ながら慎重に進まなければなりません(Fig. 3-21)。

キャラクターが移動して、抜ける床を踏んだとします(Fig. 3-22)。すると、床が抜けて落下します(Fig. 3-23)。抜けた床がどうなるかはいろいろですが、画面外まで落下して消えるか、途中で粉々に砕けるか、いずれにしても床はなくなります。

床が抜けたときに、キャラクターの移動スピードが十分に速ければ、対岸まで走り抜けることができます(Fig. 3-24)。スピードが足りないと、抜けた穴から落ちてしまいます(Fig. 3-25)。

抜ける床の見せ場は、いくつも抜ける床を横に並べたような場面です(Fig. 3-26)。この上をキャラクターが走り抜けると、キャラクターが乗った床が次々と落ちていきます(Fig. 3-27)。加速度を使って床を落とすと、いくつもの床が滑らかな放物線を描きながら落ちるため、見た目にも美しいトラップになります。

キャラクターの移動スピードが十分に速ければ、対岸まで走り抜けることができます。スピードが足りないと谷底に落ちてしまうので、一瞬たりともスピードを緩めることができない緊張感があります。特にほかのトラップや敵も出てくるときには、どのタイミングで床を一気に走り抜けたらよいのかが難しく、スリリングなアクションになります。

抜ける床はいろいろなゲームに採用されていますが、例えば「プリンス・オブ・ペルシャ」「マッピー」「ゼルダの伝説」などがあります。特に「プリンス・オブ・ペルシャ」では、床が抜けて危うく落下しそうなところを、隣の床のへりに手でつかまってよじ登る、といった緊張感あふれる演出が数多く用意されています。

**Fig. 3-21** 抜ける床

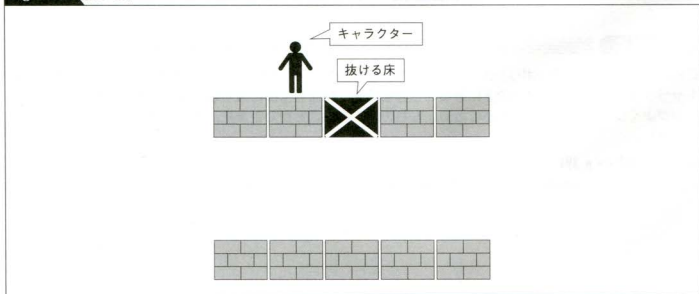



**Fig. 3-22** 抜ける床を踏む

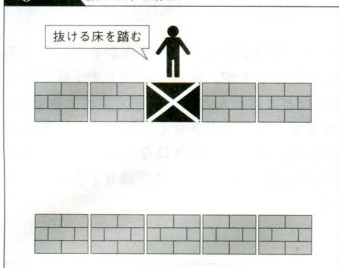



**Fig. 3-23** 床が抜けて落下する

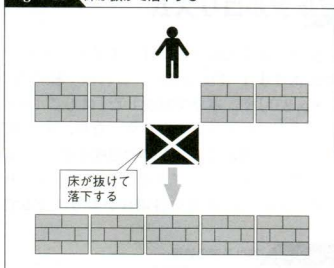



**Fig. 3-24** 対岸まで走り抜ける

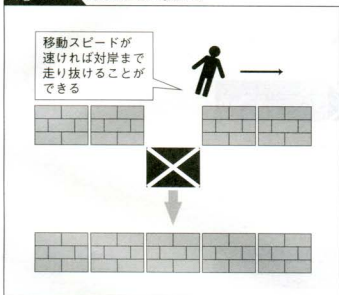



**Fig. 3-25** 抜けた穴から落ちる

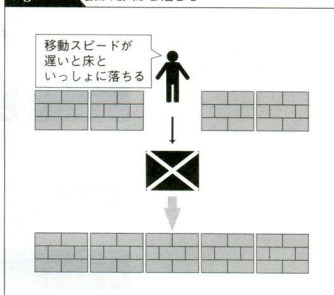



**Fig. 3-26** 並んだ抜ける床

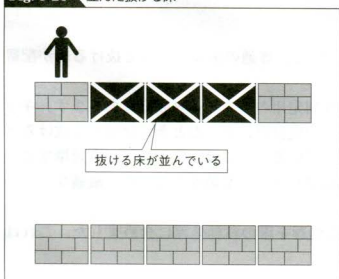
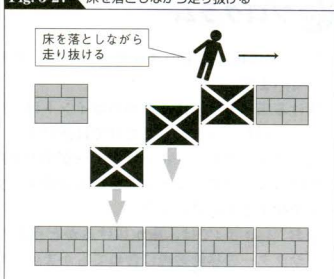
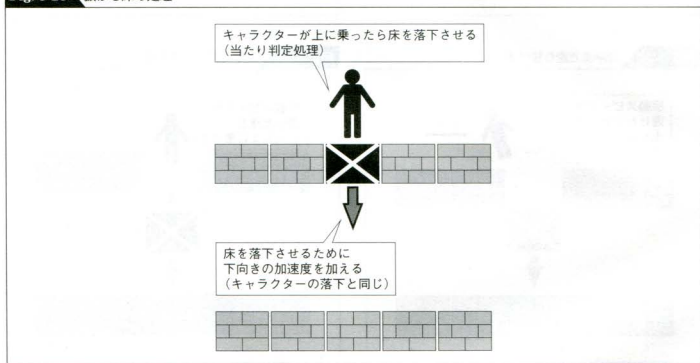



**Fig. 3-27** 床を落としながら走り抜ける

## アルゴリズム Algorithm

抜ける床を実現するには、まずキャラクターが抜ける床の上に乗ったかどうかを判定する必要があります (Fig. 3-28)。これは一種の当たり判定処理ですが、前章で解説した「飛び降り (→ p. 83)」や、本章で解説した「はしご (→ p. 128)」などとほとんど同じ処理で実現できます。

床の上にキャラクターが乗ったら、床を落下させます。床を落下させる方法はいろいろとありますが、例えば下向きの加速度を加えて落下させると、滑らかな動きになります。これはキャラクターをジャンプさせたり落下させたりする処理と同じなので、「飛び降り」をはじめとする「ジャンプ」の章の各項目が参考になるでしょう。

**Fig. 3-28** 抜ける床の処理

## プログラム Program

List 3-4は抜ける床のプログラムです。ステージには、普通の床に交じって抜ける床が配置されています。

抜ける床を上手に作るためのポイントは、当たり判定用のパラメータ設定にあります。キャラクターがちょっと乗っただけでは抜けず、しかし中央付近に乗ったときには確実に抜けるようにします。そして、キャラクターが全力で疾走しているときには、床が抜けても対岸まで走り抜けるようにすることが非常に大切です。実際にプログラムを動かしながら、最適なパラメータを探すとよいでしょう。

なお、List 3-4では、床が抜けるかどうかの判定処理を床の移動処理に含めました。これはキャラクターの移動処理に含めてもかまいません。

**List 3-4** 抜ける床(CDroppingFloorManクラス、CDroppingFloorクラス)

```
// キャラクターの移動処理を行うMove関数
bool CDroppingFloorMan::Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.2f;

    // キャラクターと床のX座標の差分の最大値
    float max_x=0.8f;

    // キャラクターと床のY座標の差分の最小値
    float min_y=0.0f;

    // キャラクターと床のY座標の差分の最大値
    float max_y=1.0f;

    // キャラクターが落下していないときの処理
    if (VY==0) {

        // レバーの入力に応じて左右の移動速度を設定する
        VX=0;
        if (is->Left) VX=-speed;
        if (is->Right) VX=speed;

        // X座標を更新し、画面端からはみ出さないように補正する
        X+=VX;
        if (X<0) X=0;
        if (X>MAX_X-1) X=MAX_X-1;
    }

    // キャラクターが床に乗っているか判定する
    // 床に乗っていないキャラクターは落下させる
    VY=speed;
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==1 &&
            abs(mover->X-X)<max_x &&
            mover->Y-Y>=min_y && mover->Y-Y<max_y
        ) {
            VY=0;
            break;
        }
    }

    // Y座標を更新する
    // キャラクターが画面下端からはみ出したときには、画面上端に戻す
    Y+=VY;
    if (Y>MAX_Y) Y=-1;
```

**List 3-4**

```
	// X方向の速度に応じて、キャラクターを傾けて表示する
	Angle=VX/speed*0.1f;

	return true;
}

// 抜ける床の移動処理を行うMove関数
bool CDroppingFloor::Move(const CInputState* is) {

	// 落下時の加速度
	float accel=0.02f;

	// キャラクターと床のX座標の差分の最大値
	float max_x=0.4f;

	// キャラクターと床のY座標の差分の最小値
	float min_y=0.0f;

	// キャラクターと床のY座標の差分の最大値
	float max_y=1.1f;

	// 床が落下していないときの処理
	// 床の上にキャラクターが乗っていたら、
	// 床を落下させる
	if (!Drop) {
		if (
			abs(X-Man->X)<max_x &&
			Y-Man->Y>min_y && Y-Man->Y<max_y
		) {
			Drop=true;
		}
	} else

	// 床が落下しているときの処理
	// 速度と座標を更新する
	// 画面端からはみ出したら、床を消去する
	// 本書のサンプルでは、Move関数でfalseを返すと、
	// 呼び出し元の関数がそのオブジェクトを消去してくれる
	{
		VY+=accel;
		Y+=VY;
		if (Y>MAX_Y) return false;
	}

	return true;
}
```

SAMPLE

「DROPPING FLOOR」は抜ける床のサンプルです。左右のレバーでキャラクターを移動させることができます。抜ける床の上にキャラクターが乗ると床が落下します。落下する前に走り抜けると、隣の床に渡ることができます。

**DROPPING FLOOR** → p. 394

# 回転ドア

キャラクターが押すと回転するドアの仕掛けです。ドアを回転させることによってステージの形が変わることを利用して、敵の追撃をかわすことができます。

ここで考える回転ドアは、中央に回転軸があり、両端に扉がついたものです (Fig. 3-29)。扉の部分を押すと、ドアを回転させることができます。

ドアを回転させるには、ドアに向かってキャラクターを移動させます。キャラクターでドアを押すと、回転ドアが回転します (Fig. 3-30)。

ドアを回転させたまま、続けてキャラクターを移動させると、ドアは90度回ったところで止まります (Fig. 3-31)。ドアの向きが変わったので、キャラクターは通過することができます。

**Fig. 3-29** 回転ドア

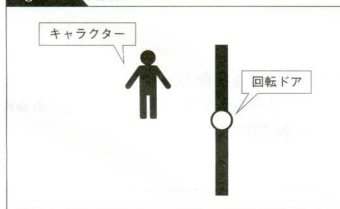



**Fig. 3-30** 回転ドアを回す

**Fig. 3-31** 回転ドアを通過する

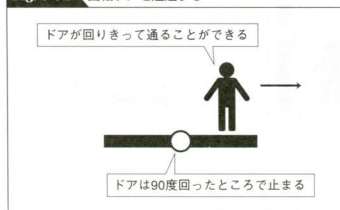



**Fig. 3-32** 横向きの回転ドアを回す

横向きになったドアも、同じように回転させることができます（Fig. 3-32）。ドアに向かってキャラクターを移動させると、ドアが回転します。縦向きのドアは左右から押しますが、横向きのドアは上下から押します。回転したドアは、90度回ったところで止まります。ドアの向きが変わるので、キャラクターはまっすぐ通過することができます。

回転ドアを採用したゲームには「レディバグ」などがあります。このゲームは「パックマン」のようにステージ内のエサを集めることが目的ですが、迷路のなかに回転ドアが設置されていることが特徴です。回転ドアを回すことによって、迷路の道筋を変えることができます。うまく道筋を変えれば、敵の追撃をかわすことができるわけです。

## アルゴリズム Algorithm

回転ドアを実現するには、まずキャラクターと回転ドアの当たり判定処理を行います。当たり判定処理そのものはほかのアクションと同様ですが、回転ドアの向きによって当たり判定が変わることに注意が必要です（Fig. 3-33）。

回転ドアが縦向きのときには、当たり判定が縦長になります。一方、回転ドアが横向きのときには、当たり判定は横長になります。このように、ドアの向きによって当たり判定を切り替えると、ドアを押すときの動きが自然になります。

回転ドアに関するもう1つのポイントはドアの回転方向です。ドアの回転方向は、次のような要素から決まります。

・ドアが縦向きか横向きか
・キャラクターがどちらの扉を押したか
・キャラクターの進行方向はどちらか

ドアが右に回転する場合（Fig. 3-34）と、ドアが左に回転する場合（Fig. 3-35）を、それぞれ図にまとめてみました。回転ドアのプログラムでは、上記のような条件を調べて、図にある8種類のどのパターンに相当するのかを判定し、ドアの回転方向を決めます。

**Fig. 3-33** ドアの向きと当たり判定

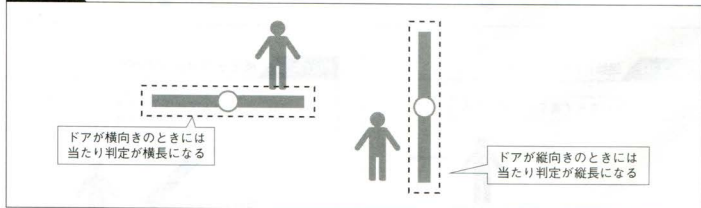



Fig. 3-34 ドアが右に回転する場合

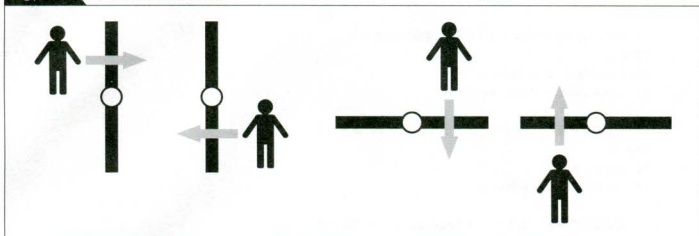
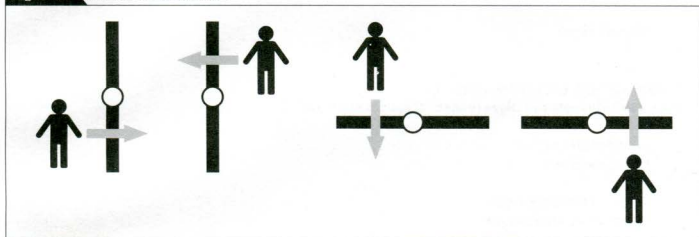



Fig. 3-35 ドアが左に回転する場合

## プログラム Program

List 3-5は回転ドアのプログラムです。キャラクターの移動処理については、レバーの入力に応じて上下左右に動くだけなので、特に変わったところはありません。回転ドアの移動処理では、キャラクターとの当たり判定処理と、ドアの回転方向を決定する処理を行います。

**List 3-5** 回転ドア(CRevolvingDoorManクラス、CRevolvingDoorクラス)

```
// キャラクターの移動処理を行うMove関数
bool CRevolvingDoorMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // レバーの入力に応じて左右の移動速度を設定する
    VX=0;
    if (is->Left) VX=-speed;
```

**List 3-5**

```
    if (is->Right) VX=speed;

    // レバーの入力に応じて上下の移動速度を設定する
    VY=0;
    if (is->Up) VY=-speed;
    if (is->Down) VY=speed;

    // X座標を更新し、画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // X座標を更新し、画面からはみ出さないように補正する
    Y+=VY;
    if (Y<0) Y=0;
    if (Y>MAX_Y-1) Y=MAX_Y-1;

    return true;
}

// 回転ドアの移動処理を行うMove関数
bool CRevolvingDoor::Move(const CInputState* is) {

    // ドアの回転スピード
    float vangle=0.02f;

    // 当たり判定の矩形の短辺
    float dist_short=1.0f;

    // 当たり判定の矩形の長辺
    float dist_long=2.0f;

    // ドアが回転していないときの処理
    // VAngleはドアの回転速度(角度の変化)を表す
    if (VAngle==0) {

        // ドアが縦向きのときに、
        // キャラクターがドアを回したかどうかを判定する
        if (
            Angle==0 &&
            abs(Man->X-X)<dist_short &&
            abs(Man->Y-Y)<dist_long
        ) {
            // キャラクターの進行方向を考慮して、回転方向を決める
            if (Man->VX<0) VAngle=-vangle;
            if (Man->VX>0) VAngle=vangle;

            // キャラクターがどちらのドアを押したかに応じて、回転方向を変える
            if (Man->Y>Y) VAngle=-VAngle;
        }
```

```
        // ドアが横向きのときに、
        // キャラクターがドアを回したかどうかを判定する
        if (
            Angle==0.25f &&
            abs(Man->X-X)<dist_long &&
            abs(Man->Y-Y)<dist_short
        ) {
            // キャラクターの進行方向を考慮して、回転方向を決める
            if (Man->VY<0) VAngle=vangle;
            if (Man->VY>0) VAngle=-vangle;

            // キャラクターがどちらのドアを押したかに応じて、回転方向を変える
            if (Man->X>X) VAngle=-VAngle;
        }
    } else

    // ドアが回転しているときの処理
    {
        // 角度の更新
        Angle+=VAngle;

        // 回転した角度の更新
        // DAngleは今回の回転で回った角度を表す
        DAngle+=abs(VAngle);

        // 回転した角度が90度を超えたら、
        // ドアの回転を止める
        // ここでは角度を0.0(0度)～1.0(360度)で表しているので、
        // 0.25は90度に相当する
        if (DAngle>=0.25f) {

            // ドアの角度を0度(縦向き)または90度(横向き)に揃える
            // これは回転時の誤差により0度や90度から微妙にずれることがあるため
            // 0.1は、0.25よりも0.0に近い値ということで適当に選んだ定数
            // 0.4は、0.25よりも0.5に近い値ということで適当に選んだ定数
            // 180度回ったドアと0度回ったドアは同じ形なので、
            // 0.5(180度)は0(0度)に補正している
            if (abs(Angle)<0.1f || abs(Angle)>0.4f) Angle=0; else Angle=0.25f;

            // 回転した角度と、回転速度を0にする
            DAngle=0;
            VAngle=0;
        }
    }

    return true;
}
```

SAMPLE

「REVOLVING DOOR」は回転ドアのサンプルです。レバーを上下左右に入力するとキャラクターを移動させることができます。回転ドアに接触すると、キャラクターの位置と進行方向に合わせてドアが回転します。

**REVOLVING DOOR** → p. 394

# ドア飛ばし

勢いよくドアを開閉して、ドアの近くにいる敵を弾き飛ばすアクションです。開いたドアを敵がくぐった瞬間に、タイミングよくボタンを押してドアを閉じることによって、敵を弾き飛ばすことができます。ドアで敵を殴るようなイメージです。

キャラクターと敵が、ドアをはさんで向かい合っているとします (Fig. 3-36)。ボタンを押すと、ドアを開閉することができます。そこで、敵を攻撃するために、あらかじめボタンを押してドアを手前に開けておきます (Fig. 3-37)。ここでは、ドアは取っ手のついている側に開くものとしました。

敵がドアをくぐる瞬間に、タイミングよくボタンを押します (Fig. 3-38)。うまくタイミングが合うと、敵はドアに弾き飛ばされてダメージを受けたり、気絶したりします (Fig. 3-39)。これがドア飛ばしです。

ドアを閉めるときには、タイミングが重要です。ドアを閉めるのが遅いと、敵がドアをくぐってきて、キャラクターはダメージを受けてしまいます (Fig. 3-40)。逆にドアを閉めるのが早いと、敵がくる前にドアが閉まってしまうので、敵を弾き飛ばすことができません (Fig. 3-41)。

なお、敵は閉まっているドアを通ることができません。ドアを早く閉めてしまったときには、いったんドアを開けて、もう一度ドア飛ばしの機会をうかがうことになります。

ドア飛ばしを採用したゲームには「マッピー」があります。このゲームでは、ドアのどちら側から敵が近づいてきたときにも、ドア飛ばしで攻撃することができます。先の解説は敵が取

**Fig. 3-36** ドアをはさんで敵と向かい合う

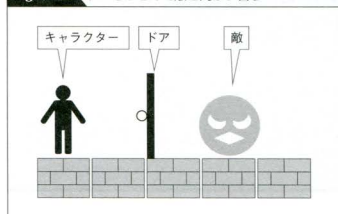



**Fig. 3-37** ドアを開けておく

Fig. 3-38 タイミングよくボタンを押す

Fig. 3-39 敵を弾き飛ばす

Fig. 3-40 ドアを閉めるのが遅かったとき

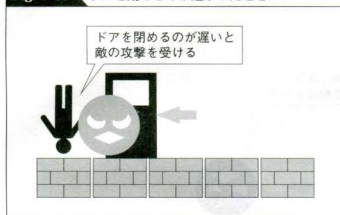



Fig. 3-41 ドアを閉めるのが早かったとき

っ手のない側から近づいてきた場合ですが、逆側から近づいてきた場合にもドア飛ばしができるようになっています。また、ドア飛ばしで敵を攻撃するだけではなく、自分のキャラクターをドアで飛ばすことでダッシュしたり、特別なドアを使って衝撃波で敵を攻撃したり、敵が自分でドアを開けたりと、ドアを使った多彩なアクションが楽しめます。

## アルゴリズム Algorithm

ドア飛ばしを実現するときのポイントは、ドアの状態を管理することです(Fig. 3-42)。ドアには次の3つの状態があります。

・閉まっている状態
・開いている状態
・閉めた直後の状態

ドアが閉まっているときにボタンを押すと、ドアが開きます。この状態でもう一度ボタンを押すと、ドアが閉まります。閉めた直後から一定時間は、ドアに攻撃判定がある状態になります。この一定時間内に敵がドアに接触すると、ドア飛ばしを行うことができます(Fig. 3-43)。

**Fig. 3-42** ドアの状態

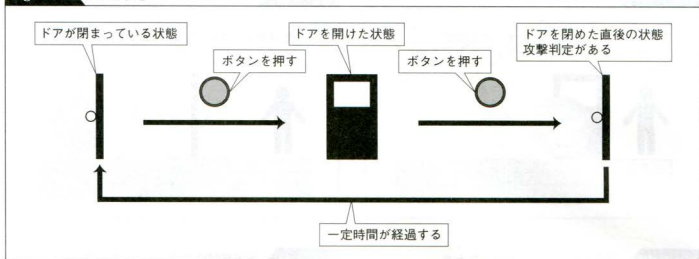
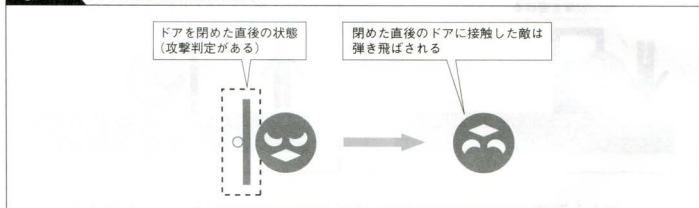



**Fig. 3-43** ドアに接触した敵を弾き飛ばす

## プログラム Program

List 3-6はドア飛ばしのプログラムです。ここにはドアと敵の移動処理を掲載しました。ドアの移動処理では、ボタンの入力に応じてドアを開閉します。閉まっている状態と、閉めた直後の状態を区別することがポイントです。また、敵の移動処理では、閉めた直後のドアに接触したときに、弾き飛ばされる処理を行います。

**List 3-6** ドア飛ばし(CDoorSmashDoorクラス、CDoorSmashEnemyクラス)

```
// ドアの移動処理を行うMove関数
bool CDoorSmashDoor::Move(const CInputState* is) {

	// 攻撃判定が出ている時間(フレーム数)
	int smash_time=10;

	// 状態に応じて分岐する
	switch (State) {
```

```
		// 閉まっている状態
		case 0:

			// 閉まったドアの画像を表示する
			Texture=Game->Texture[TEX_DOOR1];

			// ボタンを押したらドアを開ける
			if (!PrevButton && is->Button[0]) State=1;
			break;

		// 開いている状態
		case 1:

			// 開いたドアの画像を表示する
			Texture=Game->Texture[TEX_DOOR0];

			// ボタンを押したらドアを閉める
			// 攻撃判定が出ている時間を設定する
			if (!PrevButton && is->Button[0]) {
				Time=smash_time;
				State=2;
			}
			break;

		// 閉めた直後の状態
		case 2:

			// 閉まったドアの画像を表示する
			Texture=Game->Texture[TEX_DOOR1];

			// ボタンを押したらドアを開ける
			if (!PrevButton && is->Button[0]) State=1;

			// 攻撃判定が出ている残り時間を減少させ、
			// 残り時間が0になったら、
			// 閉まっている状態(攻撃判定がない)へ移行する
			Time--;
			if (Time==0) State=0;
			break;
	}

	// ボタンを押した瞬間かどうかを判定するため、
	// ボタンの入力状態を保存する
	PrevButton=is->Button[0];

	return true;
}
```

**List 3-6**

```
// 敵の移動処理を行うMove関数
bool CDoorSmashEnemy::Move(const CInputState* is) {

    // ドアと敵のX座標の差分の最大値
    // ドアとの当たり判定処理に使う
    float max_x=1.0f;

    // 弾き飛ばされたときのスピード
    float smashed_speed=0.5f;

    // 弾き飛ばされる時間(フレーム数)
    float smashed_time=10;

    // 気絶している時間(フレーム数)
    float sleep_time=60;

    // 状態に応じて分岐する
    switch (State) {

        // 通常状態
        case 0:

            // ドアが開いているか、
            // ドアが閉まっていても接触していないときには、
            // X座標を更新して左へ移動する
            // また、画面左端からはみ出したときには、
            // 画面右端から再び出現する
            if (
                Door->State==1 ||
                Door->X-X>0 || X-Door->X>max_x
            ) {
                X+=VX;
                if (X<-1) X=MAX_X;
            }

            // ドアが閉まった直後の状態(攻撃判定がある)で、
            // 敵がドアに接触したときには、弾き飛ばされる
            // 弾き飛ばされる時間を設定し、
            // 弾き飛ばされている状態へ移行する
            if (
                Door->State==2 &&
                abs(Door->X-X)<max_x
            ) {
                Time=smashed_time;
                State=1;
            }
            break;

        // 弾き飛ばされている状態
        case 1:
```

```
                // 弾き飛ばされる動きを表現するために、
                // 右方向へ勢いよく移動する
                X+=smashed_speed;

                // 一定時間が経過したら、
                // 気絶の時間を設定し、気絶状態へ移行する
                Time--;
                if (Time==0) {
                    Time=sleep_time;
                    State=2;
                }
                break;

            // 気絶状態
            case 2:

                // 一定時間が経過したら、通常状態へ戻る
                Time--;
                if (Time==0) State=0;
                break;
        }

        // 敵が気絶しているときには、
        // 画像を上下逆向きに表示する
        Angle=(State==0)?0:0.5f;

        return true;
    }
```

SAMPLE

「DOOR SMASH」はドア飛ばしのアクションのサンプルです。ボタンでドアを開閉することができます。タイミングよくドアを閉じると、敵を弾き飛ばすことができます。弾き飛ばされた敵は気絶してひっくり返ります。

**DOOR SMASH** → p. 394

# ドア閉じ込め

左右にスライドする引き戸形式のドアを開閉して、敵をドアのなかに誘い込み、ドアを閉めて敵を閉じ込めるアクションです。敵が罠にかかるのを待って閉じ込めるという手順は、「手動穴 (→ p. 226)」に似ています。

敵を誘い込むには、まずドアを開けておく必要があります (Fig. 3-44)。ドアを開くには、まずドアの取っ手がある側に移動します (Fig. 3-45)。例えば取っ手が右側についていたら、ドアの右側に移動します。

次に、左側に移動します。ドアに重なって右から左へ移動すると、キャラクターの動きに合わせて、ドアがスライドして開きます (Fig. 3-46)。

キャラクターがドアの左側に抜けると、ドアも完全に開いた状態になります (Fig. 3-47)。これで敵を誘い込むことができます。

開いたドアのなかに誘い込まれた敵は、一定時間その場で動けなくなります (Fig. 3-48)。このすきに、素早く左から右へ移動してドアを閉めます (Fig. 3-49)。開くときと同様に、ドアは

**Fig. 3-44** キャラクターとドア

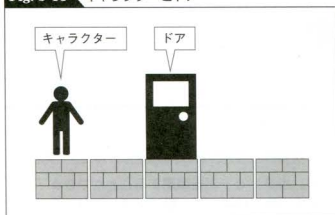



**Fig. 3-45** ドアの右側に移動する

**Fig. 3-46** ドアを開く

**Fig. 3-47** ドアが完全に開いた状態

Fig. 3-48 敵を誘い込む

Fig. 3-49 ドアを閉める

キャラクターの動きに合わせてスライドします。

敵が出てくる前にドアを完全に閉めることができれば、閉じ込めは成功です。あとは繰り返しで、再びドアの左側に移動してドアを開け、敵を誘い込む準備をします。

ドア閉じ込めを採用したゲームの例としては「ドアドア」があります。このゲームはその名のとおりドアをテーマにしたゲームです。ドアを開いて敵を誘い込み、ドアを閉じて敵を閉じ込めるという簡単なルールなのですが、ステージによってドアの配置や取っ手の方向が違っているため、ステージごとに攻略法を考える楽しみがあります。

## アルゴリズム Algorithm

ドア閉じ込めを実現するときのポイントは、ドアの開閉です(Fig. 3-50)。まず、当たり判定処理を行い、ドアにキャラクターが接触しているかどうかを調べます。接触しているときには、キャラクターが移動しているのと同じ方向にドアをスライドさせます。例えば、キャラクターの速度を使って、ドアの開閉位置を更新すればよいでしょう。

もう1つのポイントは、敵がドアに入ったときの処理です。完全に開いているドアに敵が誘

Fig. 3-50 ドアをスライドさせる

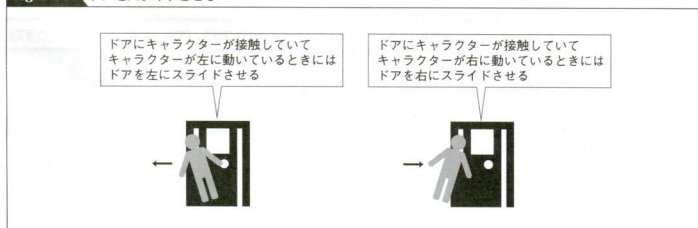
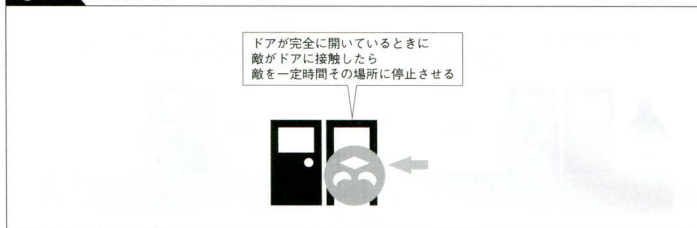



**Fig. 3-51** 敵を停止させる

い込まれたときには、敵を一定時間停止させます(Fig. 3-51)。停止している一定時間内にドアが閉められたら、その敵は消滅します。一定時間内にドアが閉められなかったら、敵は再び動き出します。

## プログラム Program

List 3-7はドア閉じ込めのプログラムです。キャラクターの移動処理、ドアの移動処理と描画処理、敵の移動処理を掲載しました。このプログラムでは敵に次の3つの状態があります。

- 通常状態：ドアに入る前の状態。左に移動する
- 停止状態：ドアに入って停止している状態
- 復帰状態：停止後に一定時間が経過して、再び移動を開始した状態。左に移動する

通常状態と復帰状態の動きは同じですが、復帰状態ではドアが開いていても停止状態にしません。これは、停止状態から復帰状態に移行したときに、再び同じドアに入って停止状態になってしまうことを防止するためです。実際のゲームでは、一定時間が経過したら復帰状態から通常状態に戻って、再びドアに誘い込まれるようにするとよいでしょう。

**List 3-7** ドア閉じ込め(CDoorConfinementManクラス、CDoorConfinementDoorクラス、CDoorConfinementEnemyクラス)

```
// キャラクターの移動処理を行うMove関数
bool CDoorConfinementMan::Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.2f;

    // レバーの入力に応じて左右方向の速度を設定する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;
```

```
    // X座標を更新し、画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    return true;
}

// ドアの移動処理を行うMove関数
bool CDoorConfinementDoor::Move(const CInputState* is) {

    // キャラクターとドアのX座標の差分の最大値
    float max_x=1.0f;

    // ドアがスライドする幅
    float slide_width=1.2f;

    // ドアにキャラクターが接触しているときには、
    // キャラクターの速度をドアの座標に加算して、
    // ドアをスライドさせる
    if (abs(Man->X-X)<max_x) X+=Man->VX;

    // 状態を0に設定する
    State=0;

    // ドアが完全に閉まっているときには、状態を1に設定する
    // OriginalXはドアがスライドして開く前のX座標
    if (X>=OriginalX) {
        X=OriginalX;
        State=1;
    }

    // ドアが完全に開いているときには、状態を2に設定する
    if (X<=OriginalX-slide_width) {
        X=OriginalX-slide_width;
        State=2;
    }
    return true;
}

// ドアの描画処理を行うDraw関数
void CDoorConfinementDoor::Draw() {

    // ドア枠を描画する
    float x=X;
    X=OriginalX;
    Texture=Game->Texture[TEX_DOOR3];
    CMover::Draw();

    // 現在のドアの位置に、ドアを描画する
```

**List 3-7**

```
    X=x;
    Texture=Game->Texture[TEX_DOOR2];
    CMover::Draw();
}

// 敵の移動処理を行うMove関数
bool CDoorConfinementEnemy::Move(const CInputState* is) {

    // ドアと敵のX座標の差分の最大値
    float max_x=0.5f;

    // ドアに入ったときに停止する時間（フレーム数）
    float stay_time=60;

    // 状態に応じて分岐する
    switch (State) {

        // 通常状態
        case 0:

            // ドアが完全に開いているときに入ったら、
            // 残り時間を設定してから、停止状態に移行する
            if (
                Door->State==2 &&
                X-Door->OriginalX>0 &&
                X-Door->OriginalX<max_x
            ) {
                Time=stay_time;
                State=1;
            }

            // X座標を更新する
            // 画面の左端から出たときには、右端に戻る
            X+=VX;
            if (X<-1) X=MAX_X;
            break;

        // 停止状態
        case 1:

            // ドアが完全に閉じられたら、敵は消滅する
            // ここでは画面の右端から再び出現させている
            if (Door->State==1) {
                X=MAX_X-1;
                State=0;
            } else

            // 残り時間を減少させる
            // 残り時間が0になったら、復帰状態に移行する
            {
```

```
                Time--;
                if (Time==0) {
                    State=2;
                }
            }
            break;

        // 復帰状態
        // 動きは通常状態と同じだが、ドアには入らない
        case 2:

            // X座標を更新する
            // 画面の左端から出たときには、右端に戻る
            X+=VX;
            if (X<-1) {
                State=0;
                X=MAX_X;
            }
            break;
    }

    // 停止状態のときには、敵の画像を上下反対に表示する
    Angle=(State==1)?0.5f:0;

    return true;
}
```

SAMPLE

「DOOR CONFINEMENT」はドア閉じ込めのアクションのサンプルです。左右のレバーでキャラクターを移動させることができます。ドアの右側（取っ手のある方）から左側に移動すると、ドアを開くことができます。左側から右側へ移動すると、ドアを閉じます。タイミングよくドアを開閉することによって、敵を閉じ込めることができます。

**DOOR CONFINEMENT** → p. 394

# エレベーター

ステージを上下する乗り物です。複数の階で構成されたステージで、上下に移動するために使います。ゲームによっては、エレベーターの屋根に乗ったり、エレベーターから飛び降りたりすることも可能です。

典型的なエレベーターは、キャラクターが乗れるかごが、天井からワイヤーで吊られている構造になっています(Fig. 3-52)。エレベーターに乗るには、キャラクターをエレベーターに重

**Fig. 3-52** エレベーター

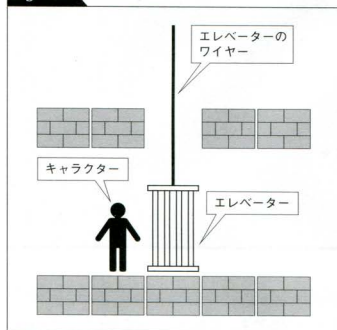



**Fig. 3-53** エレベーターに乗り込む

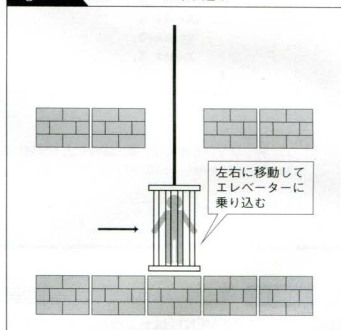



**Fig. 3-54** エレベーターで上昇する

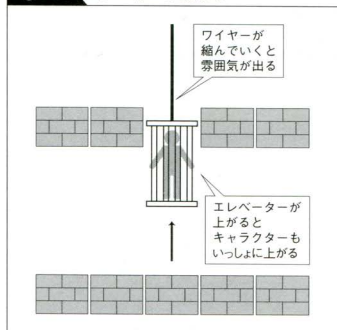



**Fig. 3-55** エレベーターから降りる

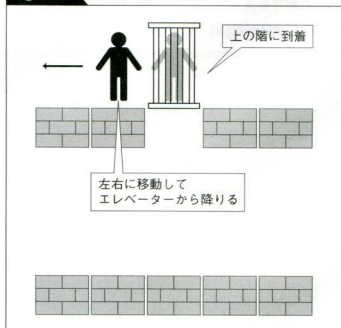



Fig. 3-56 エレベーターの穴に落ちる

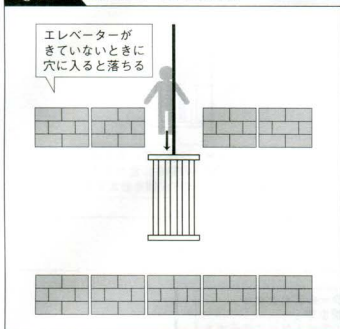
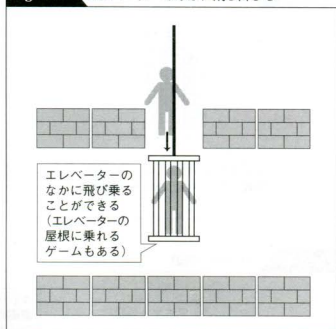



Fig. 3-57 エレベーターのなかに飛び降りる



なるまで左右に移動させます（Fig. 3-53）。

エレベーターが上昇すると、乗り込んだキャラクターもいっしょに上昇します（Fig. 3-54）。同様に、エレベーターが下降したときには、キャラクターもいっしょに下降します。エレベーターが上下に動くときには、ワイヤーも動きに合わせて伸び縮みすると、雰囲気が出ます。

上の階に到着すると、エレベーターは少しの間停止します。この間にキャラクターを左右に移動させると、エレベーターから降りることができます（Fig. 3-55）。

エレベーターがきていないときに、エレベーターの穴に入ると、キャラクターは落下します（Fig. 3-56）。これは床を踏み外して落下する場合（→ p. 83）と同じです。

落下するキャラクターの下にエレベーターがある場合には、エレベーターのなかに飛び乗ることができます。ゲームによっては、エレベーターの屋根に飛び降りることが可能なものもあります（Fig. 3-57）。

エレベーターを採用したゲームの例としては「エレベーターアクション」があります。その名のとおりエレベーターを活用したゲームですが、ほかにもエスカレーターやドアなど、さまざまな仕掛けが登場します。エレベーターに関しては、エレベーターの屋根に飛び乗ったり、エレベーターの移動方向をレバーで操作したり、あるいは敵にエレベーターを使われてしまったりと、多彩な要素が盛りこまれています。

## アルゴリズム Algorithm

エレベーターを実現するには、まずキャラクターがエレベーターに乗っているかどうかの判定処理が必要です（Fig. 3-58）。そのためには、キャラクターとエレベーターの間で当たり判定処理を行い、キャラクターがエレベーターに接触していたら、エレベーターに乗っていると判定します。エレベーターらしく見せるためには、実際にゲームを動かしながら、それらしく見

**Fig. 3-58** エレベーターに乗っているかどうかの判定

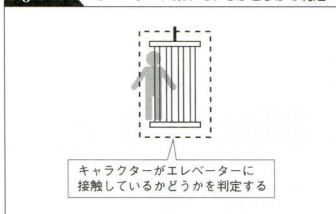



**Fig. 3-59** エレベーターで上下に移動する

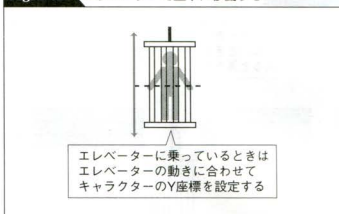
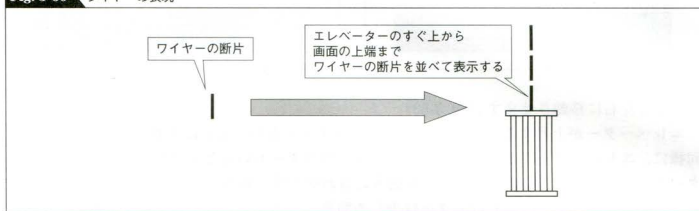



**Fig. 3-60** ワイヤーの表現

えるように当たり判定を調整する必要があります。

エレベーターに乗っているときには、エレベーターの動きに合わせて、キャラクターを上下に移動させます (Fig. 3-59)。エレベーターのY座標に合わせて、キャラクターがエレベーターに乗っている雰囲気が出るように、キャラクターのY座標を設定します。

エレベーターらしく見せるためのもう1つのポイントは、エレベーターを吊るワイヤーです (Fig. 3-60)。ワイヤーを表示するには、ワイヤーの断片の画像を用意しておきます。そして、エレベーターのすぐ上から天井や画面端まで、断片の画像を並べて表示することによって、伸び縮みするワイヤーを表現します。

## プログラム Program

List 3-8はエレベーターのプログラムです。このサンプルではエレベーターが完全に自動で動きますが、手動で動くように改造することもできます。例えば、エレベーターに乗っているときにレバーを上下に入れたら、エレベーターが指定した方向に動き出すようにします。

また、このサンプルでは上と下の2階だけですが、もっと多くの階を行き来するエレベーターを作ることも可能です。その場合には、各階のY座標をデータ化しておき、エレベーターが各階につくと止まるようにします。

**List 3-8** エレベーター(CElevatorManクラス、CElevatorクラス)

```
// キャラクターの移動処理を行うMove関数
bool CElevatorMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 床とキャラクターの当たり判定処理を行うための定数
    // X座標の差分の最大値、Y座標の差分の最小値と最大値
    float floor_max_x=0.8f;
    float floor_min_y=0.8f;
    float floor_max_y=1.0f;

    // エレベーターとキャラクターの当たり判定処理を行うための定数
    // X座標の差分の最大値、Y座標の差分の最小値と最大値
    float elevator_max_x=0.8f;
    float elevator_min_y=-0.8f;
    float elevator_max_y=-0.3f;

    // レバーの入力に応じて左右方向の速度を設定する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // Y座標の更新
    // キャラクターを落下させる
    // キャラクターがエレベーターや床に乗っている場合には、
    // この後の判定処理のなかで落下をキャンセルする
    Y+=speed;

    // エレベーターまたは床に乗っているときの処理
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();

        // エレベーターに乗っているかどうかの判定処理
        // エレベーターに乗っているときには、
        // エレベーターのY座標に応じてキャラクターのY座標を設定する
        // キャラクターはエレベーターの動きに合わせて動く
        if (
            mover->Type==2 &&
            abs(mover->X-X)<elevator_max_x &&
            mover->Y-Y>=elevator_min_y &&
            mover->Y-Y<=elevator_max_y
        ) {
```

**List 3-8**

```
            Y=mover->Y-elevator_max_y;
            break;
        }

        // 床に乗っているかどうかの判定処理
        // 床に乗っているときには、
        // 床のY座標に応じてキャラクターのY座標を設定する
        if (
            mover->Type==1 &&
            abs(mover->X-X)<floor_max_x &&
            mover->Y-Y>=floor_min_y &&
            mover->Y-Y<=floor_max_y
        ) {
            Y=mover->Y-floor_max_y;
            break;
        }
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}

// エレベーターの移動処理を行うMove関数
bool CElevator::Move(const CInputState* is) {

    // エレベーターが各階で止まる時間(フレーム数)
    int wait_time=120;

    // 上の階のY座標
    float min_y=MAX_Y-2.5f-5;

    // 下の階のY座標
    float max_y=MAX_Y-2.5f;

    // 各階で止まっているときの処理
    // 一定時間が経過したら動き出す
    if (Time>0) {
        Time--;
    } else

    // 動いているときの処理
    {
        // Y座標の更新
        Y+=VY;

        // 上の階についたら、
        // 停止する時間を設定し、
        // 進行方向を逆転する
```

```
        if (Y<=min_y) {
            Y=min_y;
            Time=wait_time;
            VY=-VY;
        }

        // 下の階についたら、
        // 停止する時間を設定し、
        // 進行方向を逆転する
        if (Y>=max_y) {
            Y=max_y;
            Time=wait_time;
            VY=-VY;
        }
    }
    return true;
}

// エレベーターの描画処理を行うDraw関数
void CElevator::Draw() {

    // エレベーターのすぐ上から画面の上端まで、
    // ワイヤーの断片を並べて表示する
    // Yは画像を表示するY座標
    // Hは画像の高さ
    float y=Y;
    Texture=Game->Texture[TEX_ELEVATOR1];
    for (Y-=H; Y+H>0; Y-=H) CMover::Draw();

    // エレベーターのかごを描画する
    Y=y;
    Texture=Game->Texture[TEX_ELEVATOR0];
    CMover::Draw();
}
```

SAMPLE

「ELEVATOR」はエレベーターのサンプルです。左右のレバーでキャラクターを移動させることができます。エレベーターに乗って、上下のフロアーに移動することができます。

**ELEVATOR** → p. 394

# 動く足場

左右や上下などに動いていて、キャラクターが乗ることができる仕掛けです。ゲームによって、動く床だったり動くブロックだったりします。あるいは、雲や動物といった場合もあります。

動く足場の多くは、崖に設置されています (Fig. 3-61)。動く足場をうまく使わないと、対岸に渡ることができません。

ほとんどの場合、足場は一定の範囲を往復しています。移動範囲の端までくると、多くのゲームでは足場が少しの間停止します (Fig. 3-62)。この停止しているすきを狙って、キャラクターを左右に移動させ、素早く足場に乗ります (Fig. 3-63)。

うまく足場に乗ることができると、足場に乗ったまま移動することができます (Fig. 3-64)。足場の上でレバーを入力せずに止まっているときには、キャラクターは足場と同じ速度で移動します。レバーを入力して、足場の上で左右に動くこともできます。

**Fig. 3-61** 動く足場

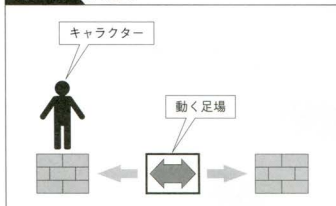



**Fig. 3-62** 停止する足場

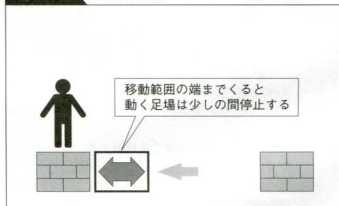



**Fig. 3-63** 足場に乗る

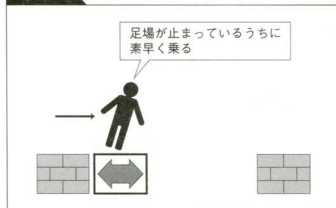



**Fig. 3-64** 足場に乗って移動する

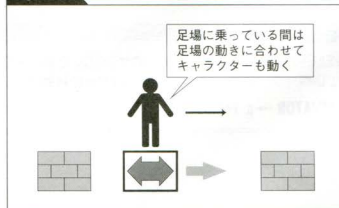
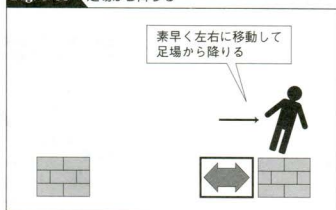



Fig. 3-65 対岸に到着

Fig. 3-66 足場から降りる

対岸につくと、足場はまた少しの間停止します(Fig. 3-65)。このタイミングを狙って、キャラクターを移動させて足場から降ります(Fig. 3-66)。これで対岸に無事渡ることができました。

動く足場を採用したゲームは非常に数多くあります。例えば「スーパーマリオブラザーズ」にも、動く足場が登場します。動く足場が登場するステージには、たいてい崖や池といった、キャラクターが落ちるとミスになる地形があります。移動やジャンプを駆使して、足場を上手に渡り、落下せずに進むことがこういったステージの目的です。

## アルゴリズム Algorithm

動く足場を実現するには、まずキャラクターが足場に乗っているかどうかの判定処理が必要です(Fig. 3-67)。これは床に乗っているかどうかの判定処理とまったく同じ処理になります。当たり判定の大きさは、足場に乗っている雰囲気が出るように、実際にゲームを動かしながら調整するとよいでしょう。

判定処理の結果、足場に乗っていると判定されたら、足場に合わせてキャラクターを動かします(Fig. 3-68)。左右に動く足場の場合には、足場のX方向の速度をキャラクターのX座標に加算します。足場の速度をfvx、キャラクターの座標をxとすると、

```
x+=fvx
```

となります。

Fig. 3-67 足場に乗っているかどうかの判定

足場に乗っているかどうかの判定

床に乗っているかどうかの判定と同じ

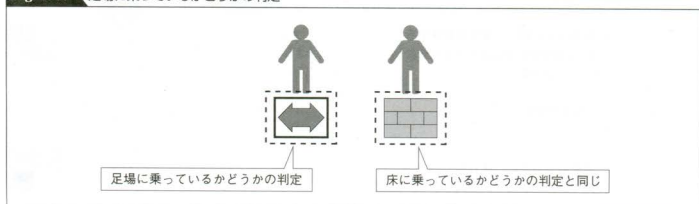
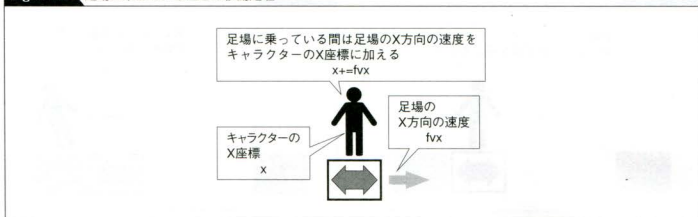



**Fig. 3-68** 足場に乗っているときの移動処理

## プログラム Program

List 3-9は動く足場のプログラムです。レバーを入力したときには、足場の速度とは別に、レバーによる速度をキャラクターの座標に加算します。こうすると、足場に乗っているときに、足場の上でキャラクターを左右に動かすことができます。

**List 3-9** 動く足場（CMovingFloorクラス、CMovingFloorManクラス）

```
// 足場の移動処理を行うMove関数
bool CMovingFloor::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.1f;

    // 移動範囲の両端で停止する時間（フレーム数）
    int wait_time=60;

    // 停止中の処理
    // 残り時間を減少させる
    if (Time>0) {
        Time--;
    } else

    // 移動中の処理
    {
        // 移動方向に応じて速度を計算する
        // Dirは移動方向を-1または1で表す
        VX=Dir*speed;

        // X座標の更新
        X+=VX;

        // 移動範囲の両端に達したら、
        // 停止時間を設定し、移動速度を0にして、
```

```
            // 移動方向を逆転させる
            if (
                X<=OriginalX ||
                X>=OriginalX+Range
            ) {
                Time=wait_time;
                VX=0;
                Dir=-Dir;
            }
        }

        return true;
}

// キャラクターの移動処理を行うMove関数
bool CMovingFloorMan::Move(const CInputState* is) {

        // 移動スピード
        float speed=0.2f;

        // 足場や床との当たり判定処理を行うための定数
        // X座標の差分の最大値、Y座標の差分の最小値と最大値
        float max_x=0.8f;
        float min_y=0.8f;
        float max_y=1.0f;

        // レバーの入力に応じて左右方向の速度を設定する
        VX=0;
        if (is->Left) VX=-speed;
        if (is->Right) VX=speed;

        // X座標を更新し、キャラクターが画面からはみ出さないように補正する
        X+=VX;
        if (X<0) X=0;
        if (X>MAX_X-1) X=MAX_X-1;

        // Y座標を更新する
        // キャラクターが画面の下端からはみ出したときには、画面の上端から出現させる
        Y+=speed;
        if (Y>MAX_Y) Y=-1;

        // 足場や床に乗っているかどうかの判定処理
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (
                mover->Type!=0 &&
                abs(mover->X-X)<max_x &&
                mover->Y-Y>=min_y &&
                mover->Y-Y<=max_y
            ) {
                // 足場や床にキャラクターがちょうど乗るように、
```

**List 3-9**

```
				// Y座標を調整する
				Y=mover->Y-max_y;

				// 足場に乗っている場合には、
				// キャラクターのX座標に足場のX方向の速度を加算して、
				// キャラクターを足場といっしょに動かす
				if (mover->Type==2) {
					CMovingFloor* floor=(CMovingFloor*)mover;
					X+=floor->VX;
					break;
				}
			}
		}

		// X方向の速度に応じて、キャラクターを傾けて表示する
		Angle=VX/speed*0.1f;

		return true;
}
```

SAMPLE

「MOVING FLOOR」は動く足場のサンプルです。左右のレバーでキャラクターを移動させることができます。左右に移動する床の上にタイミングよくキャラクターを乗せると、対岸に渡ることができます。

**MOVING FLOOR** → p. 395

# ベルトコンベア

上に乗ると、自動的にキャラクターが左右へ運ばれていく仕掛けです。現実世界のベルトコンベアは荷物やセメントなどを運ぶために使いますが、ゲームの世界ではキャラクターが乗るためのものとして多く使われています。ゲームによっては、ベルトコンベアに乗って敵や障害物が運ばれてくることもあります。

ベルトコンベアには決まった移動方向があり、上に乗ったキャラクターはその方向へ運ばれていきます。ベルトコンベアの方向が一定時間ごとに変わることもあります。

例えば、ベルトコンベアはFig. 3-69のように表現できます。多くのゲームでは、ベルトを動かしたり、ベルトを回す軸を回転させたりして、動いている雰囲気を出します。ベルトコンベアはどちらの方向に動いているのかを見極めるのが重要なので、ベルトや軸の動きはプレイヤーにとっても重要な情報です。

ベルトコンベアに乗ると、キャラクターはベルトコンベアの移動方向に運ばれていきます(Fig. 3-70)。そして、端まで運ばれると、キャラクターは自動的にベルトコンベアから降りま

**Fig. 3-69** ベルトコンベア

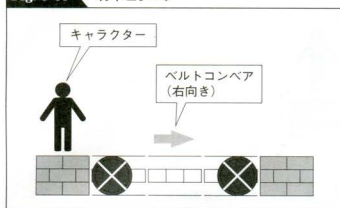



**Fig. 3-70** ベルトコンベアを使って移動する

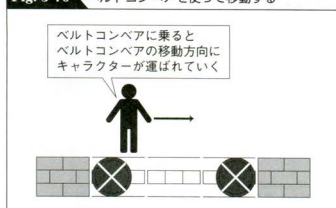



**Fig. 3-71** ベルトコンベアから降りる

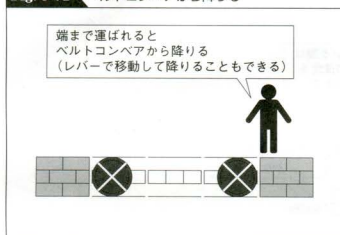
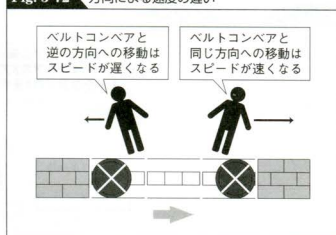



**Fig. 3-72** 方向による速度の違い

す (Fig. 3-71)。レバーで左右に移動して、ベルトコンベアから降りることもできます。

ベルトコンベアの上でレバーを使って移動するときには、移動方向によってスピードが変わります。ベルトコンベアの移動方向と同じ方向への移動はスピードが速くなり、逆の方向に移動しようとするとスピードが遅くなります (Fig. 3-72)。

ベルトコンベアを採用したゲームも数多くあります。例えば「ザ・キャッスル」には、ベルトコンベアが多用されています。ベルトコンベアで自分のキャラクターが移動するだけではなく、金庫やブロックといったものもベルトコンベアに乗って流れてきます。これを利用して、遠い場所から必要な場所へものを移動させることができます。このゲームには、こういったベルトコンベアを使った多様なパズルが盛り込まれています。

## アルゴリズム Algorithm

ベルトコンベアの実現方法は、「動く足場 (→ p. 166)」に似ています。まず、キャラクターがベルトコンベアに乗っているかどうかの判定が必要です (Fig. 3-73)。これは、床に乗っているかどうかの判定処理と同じ処理です。

ベルトコンベアに乗っていると判定されたら、ベルトコンベアの移動方向に合わせて、キャラクターを動かします (Fig. 3-74)。それには、ベルトコンベアの移動速度を、キャラクターの

**Fig. 3-73** ベルトコンベアに乗っているかどうかの判定

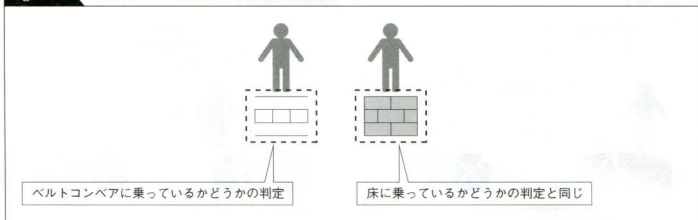
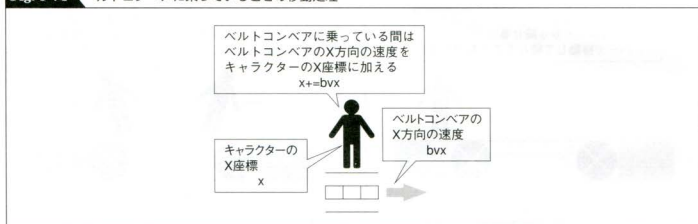



**Fig. 3-74** ベルトコンベアに乗っているときの移動処理

X座標に加算します。ベルトコンベアの速度をbvx、キャラクターの座標をxとすると、

```
x+=bvx
```

となります。

レバーの入力による速度をvxとすると、キャラクターの移動速度は、

```
bvx+vx
```

のように、レバー入力による速度とベルトコンベアの速度の合計になります。ベルトコンベアの向きとレバーの向きが同じならば、vxとbvxの符号が同じになるため、速度の絶対値が大きくなります。つまり、ベルトコンベアと同じ向きに移動するときには、移動スピードが大きくなるというわけです。逆の向きに移動するときには、レバーによる速度とベルトコンベアの速度が打ち消し合うため、移動スピードは小さくなります。

## プログラム Program

List 3-10はベルトコンベアのプログラムです。実はベルトコンベアは、「動く足場(→ p. 166)」とほとんど同じ処理で実現できます。動く足場のプログラム (List 3-9) と見比べてみてください。

**List 3-10** ベルトコンベア(CConveyorManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // ベルトコンベアや床との当たり判定処理を行うための定数
    // X座標の差分の最大値、Y座標の差分の最小値と最大値
    float max_x=0.8f;
    float min_y=0.8f;
    float max_y=1.0f;

    // レバーの入力に応じて左右方向の速度を設定する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // Y座標を更新する
    // 画面の下端からはみ出したときには、画面の上端から出現させる
    Y+=speed;
    if (Y>MAX_Y) Y=-1;

    // ベルトコンベアや床に乗っているかどうかの判定処理
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type!=0 &&
            abs(mover->X-X)<max_x &&
            mover->Y-Y>=min_y &&
            mover->Y-Y<=max_y
        ) {
            // ベルトコンベアや床にキャラクターがちょうど乗るように、
            // Y座標を調整する
            Y=mover->Y-max_y;

            // ベルトコンベアに乗っている場合には、
            // キャラクターのX座標にベルトコンベアの移動速度を加算して、
            // キャラクターを運ぶ
            if (mover->Type==2) {
                CConveyor* conveyor=(CConveyor*)mover;
                X+=conveyor->VX;
            }
        }
```

**List 3-10**

```
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

**SAMPLE**

「BELT CONVEYOR」はベルトコンベアのサンプルです。左右のレバーでキャラクターを移動させることができます。ベルトコンベアの上にキャラクターが乗ると、キャラクターはベルトコンベアの進行方向に運ばれていきます。また、ベルトコンベアの上をキャラクターが移動する際には、進行方向に移動する場合はベルトコンベアによって加速し、反対方向に移動する場合は減速します。

**BELT CONVEYOR** → p. 395

## 上昇気流

空中で気流に乗ることで、キャラクターが浮かび上がる仕掛けです。ゲームによっては、地面に扇風機のような装置が置かれていて、そこから上昇気流が発生している場合もあります。また、扇風機が横に配置されていて、キャラクターが横に飛ばされたり、上に配置された扇風機に押し下げられたりと、いろいろなバリエーションがあります。

例えば、扇風機のような装置を配置し、その上方に上昇気流が発生するようにします(Fig. 3-75)。気流が上に移動するグラフィックなどを表示して、上昇気流がどこに発生しているのかを表現すると、プレイしやすくなります。

**Fig. 3-75** 上昇気流

キャラクター
空中で左右に動ける
上昇気流を表現するため
グラフィックを上に移動させる
上昇気流
扇風機など

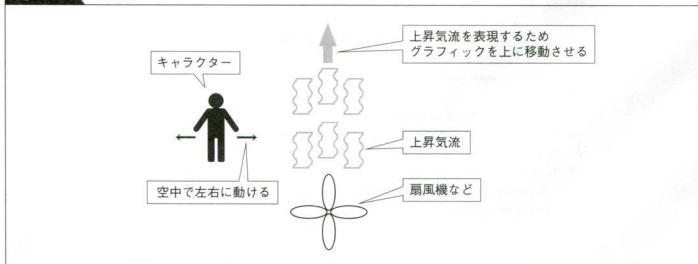
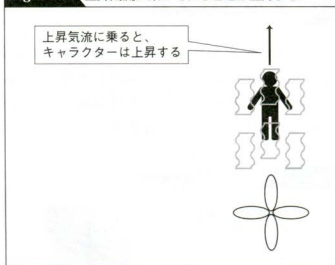



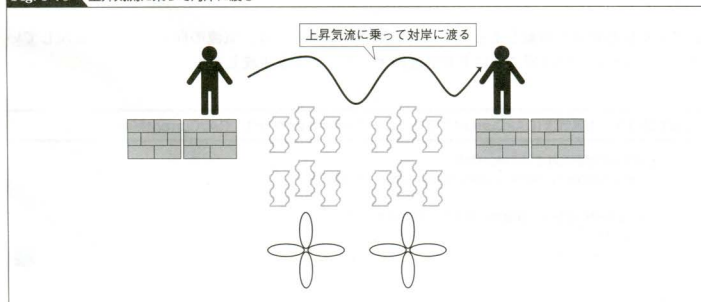
Fig. 3-76 上昇気流に乗っていないときは落下する

Fig. 3-77 上昇気流に乗っているときは上昇する

上昇気流を使うゲームでは、キャラクターが空中でもある程度左右に動けることが一般的です。例えば、キャラクターがグライダーで飛行しているような状況です。空中で左右に動き、上昇気流をつかまえて上昇することが、ゲームの目的になります。

上昇気流に乗っていないときには、キャラクターは落下します (Fig. 3-76)。上昇気流に乗ると、上昇することができます (Fig. 3-77)。

上昇気流を使ったステージとしては、Fig. 3-78のようなものをよく見かけます。広い谷間があって、谷底から上昇気流が発生しています。対岸に渡るためには、上昇気流を上手につかまえて、谷底に落ちないようにキャラクターを飛行させる必要があります。

上昇気流にかぎらず、気流や扇風機を使ったゲームはいろいろとあります。例えば3Dゲームですが、「ゼルダの伝説 風のタクト」は気流を使ったシーンが数多く出てくるゲームです。上昇気流に乗るだけではなく、タクト (指揮棒) を使って風向きをコントロールすることもできます。上昇気流と風向きを組み合わせて、遠く離れた場所に飛んでいったり、空中にある取りにくいアイテムを取ったりといった、謎解き要素を含んだアクションが楽しめます。

Fig. 3-78 上昇気流に乗って対岸に渡る

## アルゴリズム Algorithm

上昇気流を実現するには、上昇気流の有効範囲内にキャラクターが入っているかどうかを判定します（Fig. 3-79）。上昇気流の有効範囲は、扇風機などの上空に広がっています。この範囲にキャラクターが入ると、上昇気流の力によって、キャラクターは上昇します。有効範囲から外れると、キャラクターは落下します（Fig. 3-80）。

**Fig. 3-79** 上昇気流の有効範囲内にいるとき

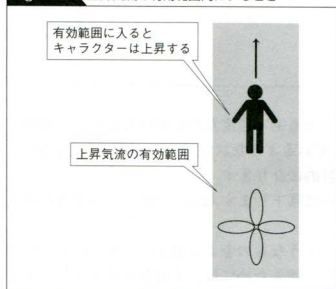
有効範囲に入ると
キャラクターは上昇する

上昇気流の有効範囲

**Fig. 3-80** 上昇気流の有効範囲外にいるとき

## プログラム Program

List 3-11は上昇気流のプログラムです。ここでは、

- 上昇気流
- 扇風機
- キャラクター

に関する移動処理を掲載しました。上昇気流オブジェクトは、気流の向きと位置を表現しています。扇風機は一定時間ごとに上昇気流のオブジェクトを生成します。

**List 3-11** 上昇気流（CAirCurrentクラス、CAirCurrentFanクラス、CAirCurrentManクラス）

```
// 上昇気流の移動処理を行うMove関数
bool CAirCurrent::Move(const CInputState* is) {

    // 上方向に移動し、画面端に隠れたら消滅する
    Y-=0.1f;
    return Y>=-1;
```

```
}

// 扇風機の移動処理を行うMove関数
bool CAirCurrent::Move(const CInputState* is) {

    // 上昇気流のオブジェクトを発生させる間隔（フレーム数）
    int wait_time=40;

    // 扇風機が回っている様子を表現するため、
    // 画像を回転させて表示する
    Angle+=0.01f;

    // 次の上昇気流を発生させるまでの残り時間を減少させる
    if (Time>0) {
        Time--;
    } else

    // 残り時間が0になったときには、
    // 上昇気流のオブジェクトを生成し、
    // 次に生成するまでの残り時間を設定する
    {
        new CAirCurrent(X, Y);
        Time=wait_time;
    }

    return true;
}

// キャラクターの移動処理を行うMove関数
bool CAirCurrent::Move(const CInputState* is) {

    // 左右方向の移動スピード
    float speed=0.2f;

    // 上昇スピード
    float up_speed=-0.05f;

    // 落下スピード
    float down_speed=0.03f;

    // 上昇気流との当たり判定処理を行うための定数
    // X座標の差分の最大値
    float max_x=0.6f;

    // レバーの入力に応じて左右方向の速度を設定する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、画面からはみ出さないように補正する
```

**List 3-11**

```
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // 上昇気流の有効範囲内のときには上向きの速度を、
    // 有効範囲外のときには下向きの速度を設定する
    float vy=down_speed;
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==1 &&
            abs(mover->X-X)<max_x
        ) {
            vy=up_speed;
            break;
        }
    }

    // Y座標を更新し、キャラクターが画面からはみ出さないように補正する
    Y+=vy;
    if (Y<0) Y=0;
    if (Y>MAX_Y-1) Y=MAX_Y-1;

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「AIR CURRENT」は上昇気流のサンプルです。左右のレバーでキャラクターを移動させることができます。気流に乗るとキャラクターは上昇します。気流から外れると、キャラクターはゆっくりと落下します。

**AIR CURRENT** → p. 395

# ワープゲート

なかに入ると、キャラクターが別の場所に瞬間移動する仕掛けです。ゲームによってワープゲートの形はさまざまで、ゲートの形をしていたり、ドアだったり、あるいは土管だったりします。多くのゲームでは、ワープゲートに入ったキャラクターは、別のワープゲートから出てきます。移動先はあらかじめ決まっていることがほとんどで、入るたびに移動先がランダムに変わるものはまれです。

例えば、Fig. 3-81のような土管状のワープゲートを考えてみましょう。これは上から土管に入ることによって、別の土管からキャラクターが出てくるタイプのワープゲートです。

ここではレバー操作でワープゲートに入ることにしましょう。ワープゲートの上でレバーを下に入力すると、キャラクターがゲートに入ります(Fig. 3-82)。キャラクターは、いったん完全にゲートに入って見えなくなったあと、別のゲートから出現します(Fig. 3-83)。キャラクターがゲートから上がってきて、完全にゲートの外に出たらワープは完了です(Fig. 3-84)。

**Fig. 3-81** ワープゲート

キャラクター
ワープゲート
（土管）
別の場所にある
ワープゲート

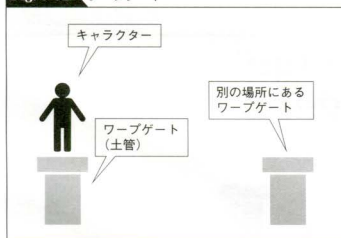



**Fig. 3-82** ワープゲートに入る

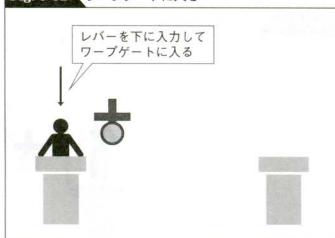



**Fig. 3-83** 別のワープゲートから出てくる

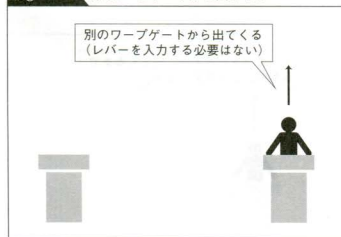



**Fig. 3-84** ワープの完了

ワープゲートはいろいろなゲームで採用されています。例えば「スーパーマリオブラザーズ」では、土管状のワープゲートが登場します。このゲームにはステージ内に多数の土管が配置されていますが、ワープできる土管もあれば、何の仕掛けもない土管もあります。また、ボーナスステージにいける土管や、敵が出てくる土管などもあります。一度使ってみるまで何が起こるかわからない、スリルのある仕掛けだといえます。

## アルゴリズム Algorithm

ワープゲートを実現するには、まずキャラクターがワープゲートの上に乗っているかどうかを判定します(Fig. 3-85)。これは一種の当たり判定処理です。キャラクターの座標がワープゲート上の一定範囲内にあれば、ワープゲートに乗っているということです。ワープゲートに乗った状態で、レバーを下に入力していたら、ワープゲートの動作を開始します。

ワープを開始したら、キャラクターをワープゲートのなかに吸い込みます。これはレバー操作ではなく、自動的にキャラクターを下降させるとよいでしょう。最初のゲートにキャラクターが完全に隠れたら、今度は別のゲートからキャラクターを出現させます。この場合もキャラクターを自動的に上昇させて、ゲートの外に完全に出るまで動かします(Fig. 3-86)。

**Fig. 3-85** ワープの開始条件

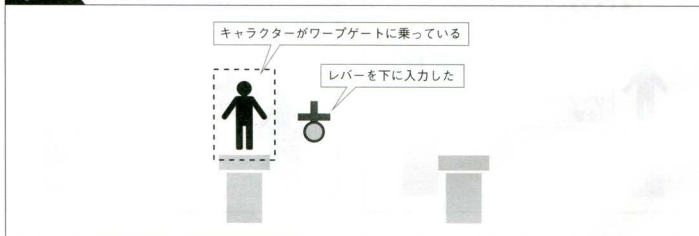
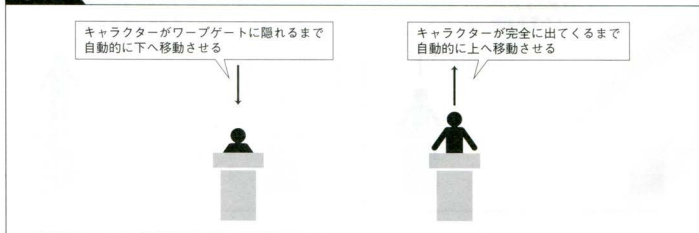



**Fig. 3-86** ワープゲートの出入りの動作

## プログラム Program

List 3-12はワープゲートのプログラムです。このサンプルでは、入ったワープゲートの5個右にあるゲートがワープの出口になっています。出口のワープゲートの座標は、入口となるゲートのオブジェクトが保持しています。ワープ時にX座標だけではなくY座標も設定するようにすれば、横方向に移動するだけではなく、自由な座標にワープすることができます。

**List 3-12** ワープゲート(CWarpGateManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 左右方向の移動スピード
    float speed=0.1f;

    // ワープゲートに出入りするときの上下方向のスピード
    float warp_speed=0.1f;

    // ワープゲートとの当たり判定処理を行うための定数
    // X座標の差分の最大値
    float max_x=0.5f;

    // 状態に応じて分岐する
    switch (State) {

        // 通常状態
        case 0:

            // レバーの入力に応じて左右方向の速度を設定する
            VX=0;
            if (is->Left) VX=-speed;
            if (is->Right) VX=speed;

            // X座標を更新し、キャラクターが画面からはみ出さないように補正する
            X+=VX;
            if (X<0) X=0;
            if (X>MAX_X-1) X=MAX_X-1;

            // ワープゲートに入ったかどうかの判定処理
            if (is->Down) {
                for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
                    CMover* mover=(CMover*)i.Next();
                    if (
                        mover->Type==1 &&
                        abs(mover->X-X)<max_x
                    ) {
```

**List 3-12**

```
                    // レバーを下に入力していて、
                    // かつワープゲートに乗っている場合には、
                    // ワープゲートに入る
                    // 目的地のX座標(DestX)を設定し、
                    // ワープゲートに入っていく状態へ移行する
                    CWarpGate* gate=(CWarpGate*)mover;
                    DestX=gate->DestX;
                    State=1;
                    break;
                }
            }
        }
        break;

    // ワープゲートに入っていく状態
    case 1:

        // Y座標を更新して、下に移動する
        Y+=warp_speed;

        // キャラクターがゲートへ完全に入ったら、
        // X座標を目的地のX座標にして、
        // ワープゲートから出てくる状態へ移行する
        if (Y>=MAX_Y-1) {
            X=DestX;
            State=2;
        }
        break;

    // ワープゲートから出てくる状態
    case 2:

        // Y座標を更新して、上に移動する
        Y-=warp_speed;

        // キャラクターがゲートから完全に出たら、
        // 通常状態へ戻る
        if (Y<=MAX_Y-2) State=0;
        break;
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「WARP GATE」はワープゲートのサンプルです。ワープゲート(土管)の上でレバーを下に入れるとキャラクターがワープして、別のゲートから出てきます。

**WARP GATE** → p. 395

# 切り替わる通路

キャラクターが通路を押したり、ボタンを入力したりすることによって、通路の形が切り替わる仕掛けです。通路を切り替えることによって、切り替える前には進めなかった場所にいけるようになります。

ここでは、はしごの形を切り替える例を説明しましょう。切り替えを行うには、まずキャラクターがはしごに接触する必要があります(Fig. 3-87)。

はしごに接触した状態でボタンを押すと、はしごを掛け替えることができます(Fig. 3-88)。はしごを構成するパーツが左右にスライドして、右上の床に掛かっていたはしごが、左上の床に掛かります。再びはしごに接触した状態でボタンを押すと、はしごは元の位置に戻ります。このようにはしごを掛け替えることによって、右上の床にいったり、左上の床にいったりと、行き先を変えることができます。

切り替わる通路は「Mr.Do! キャッスル」などに採用されています。このゲームでは、上下の階をつなぐ階段をキャラクターが押すことによって、階段を左右にスライドさせ、掛け替えることができます。階段の切り替えを利用して、敵から逃げたり、アイテムを集めたりします。

**Fig. 3-87** 切り替わるはしご

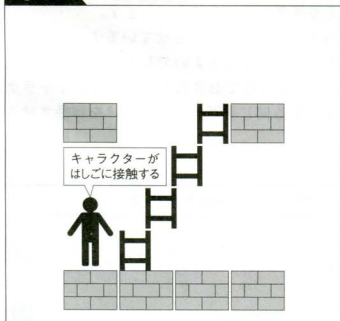
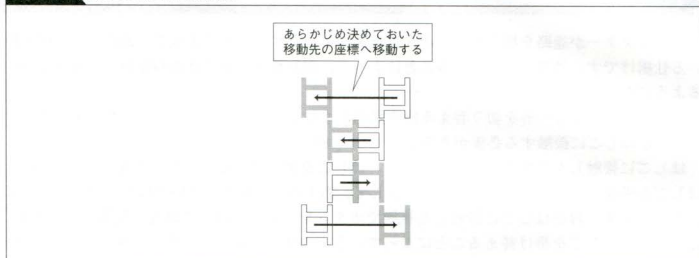



**Fig. 3-88** はしごを切り替えて別の形にする

## アルゴリズム Algorithm

切り替わる通路を実現するときのポイントは、通路の切り替え方法です（Fig. 3-89）。はしごや階段といった通路を構成するパーツごとに、切り替えたあとの移動先をあらかじめ決めて、座標をデータ化しておきます。そして、キャラクターが通路を押したり、通路に接触してボタンを入力したりしたら、各パーツを用意しておいた座標に移動させます。

**Fig. 3-89** 切り替わる通路の実現方法

## プログラム Program

List 3-13は切り替わる通路のプログラムです。キャラクターの移動処理については、「はしご（→ p. 128）」とほぼ同じです。こちらのプログラムでは、床に乗っているかどうかを、当たり判定処理を使って調べているので、床が複数階あるステージにも対応できます。

はしごの切り替えについては、画面上にあるすべてのパーツを移動させています。実際のゲームでは、1つのはしごを構成しているパーツだけを移動させるとよいでしょう。これを実現するには、1つのはしごを構成するパーツに共通の番号を付けておきます。そして、キャラクターが接触しているパーツの番号を調べて、画面上にある同じ番号のパーツだけを移動させるようにします。

**List 3-13** 切り替わる通路（CSwitchingPathクラス、CSwitchingPathManクラス）

```
// 切り替わるはしごの移動処理を行うMove関数
// 通路の切り替え中ならば、はしごのパーツを移動させる
// 移動先の座標に達したら、移動を終了する
// Leftは左側の移動先座標
// Rightは右側の移動先座標
```

```
bool CSwitchingPath::Move(const CInputState* is) {
    if (Switching) {
        X+=VX;
        if (X<=Left || X>=Right) Switching=false;
    }
    return true;
}

// 通路を切り替えたいときに呼び出すSwitch関数
// 通路の切り替え中でなければ、切り替えを開始する
// 移動先の座標に応じて、移動速度を設定する
void CSwitchingPath::Switch() {
    if (!Switching) {
        float speed=0.1f;
        if (X>Left) VX=-speed; else VX=speed;
        Switching=true;
    }
}

// キャラクターの移動処理を行うMove関数
bool CSwitchingPathMan::Move(const CInputState* is) {

    // 左右方向の移動スピード
    float speed=0.2f;

    // はしごとの当たり判定処理を行うための定数
    // X座標の差分の最大値、Y座標の差分の最小値と最大値
    float ladder_max_x=0.6f;
    float ladder_min_y=-1.0f;
    float ladder_max_y=1.0f;

    // 床との当たり判定処理を行うための定数
    // X座標の差分の最大値、Y座標の差分の最小値と最大値
    float floor_max_x=0.8f;
    float floor_min_y=0.6f;
    float floor_max_y=1.0f;

    // はしごに接触しているかどうか
    bool ladder=false;

    // 床に乗っているかどうか
    bool floor=false;

    // はしごにつかまっているかどうかの判定処理
    // はしごとキャラクターの当たり判定処理を行う
    // つかまっているときにはladderをtrueにする
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==2 &&
```

## List 3-13

```
        abs(mover->X-X)<ladder_max_x &&
        mover->Y-Y>=ladder_min_y &&
        mover->Y-Y<=ladder_max_y
    ) {
        // 単にladderをtrueにすると、
        // はしごの上端でキャラクターが振動してしまうので、
        // はしごの上端付近にいるときには床に乗っている扱いにしている
        if (mover->Y-Y<ladder_max_y-speed) ladder=true; else floor=true;
        break;
    }
}

// 床に乗っているかどうかの判定処理
// 床とキャラクターの当たり判定処理を行う
// 乗っているときにはfloorをtrueにする
for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
    CMover* mover=(CMover*)i.Next();
    if (
        mover->Type==1 &&
        abs(mover->X-X)<floor_max_x &&
        mover->Y-Y>=floor_min_y &&
        mover->Y-Y<=floor_max_y
    ) {
        floor=true;
        break;
    }
}

// はしごにつかまっているか、床に乗っているときの処理
if (ladder || floor) {

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // レバーの入力に応じて上下に移動する
    VY=0;
    if (ladder && is->Up) VY=-speed;
    if (is->Down) VY=speed;

    // Y座標を更新し、キャラクターが画面からはみ出さないように補正する
    Y+=VY;
    if (Y<0) Y=0;
    if (Y>ground_y) Y=ground_y;
```

```
        // はしごに接触しながら、ボタンを押したときの処理
        // はしごの切り替えを行う
        // はしごを構成するすべてのパーツについて、
        // Switch関数を呼び出し、位置の移動を行う
        if (ladder && is->Button[0]) {
            for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
                CMover* mover=(CMover*)i.Next();
                if (mover->Type==2) {
                    CSwitchingPath* ladder=(CSwitchingPath*)mover;
                    ladder->Switch();
                }
            }
        }
    }

    // はしごにつかまっておらず、床にも乗っていないとき、
    // すなわち落下しているときの処理
    // レバーによる操作は不可能で、ただ落下する
    if (!ladder && !floor) {
        Y+=speed;
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「SWITCHING PATH」は切り替わる通路のサンプルです。左右のレバーでキャラクターを移動させることができます。はしごに乗っているときは、上下のレバーではしごを昇降することができます。はしごに接している状態でボタンを押せば、はしごの向きを切り替えることができます。

**SWITCHING PATH** → p. 395

# 壁を壊して通路を作る

壁やブロックに行く手を阻まれているときに、壁を壊して通路を作るアクションです。壁をキャラクターが直接壊す場合もあれば、爆弾などのアイテムを使って壊す場合もあります。

壁に道をふさがれているような状況を考えてみましょう（Fig. 3-90）。壁がある場所は通ることができません。ボタンを押したり、アイテムを使ったりすると、周囲の壁を壊すことができます（Fig. 3-91）。移動と壁の破壊を繰り返すことによって、壁を壊して道を切り開きながら進むことが可能です（Fig. 3-92）。

壁を壊して通路を作るゲームには、例えば「ちゃっくんぽっぷ」があります。このゲームには壊せる壁と壊せない壁があります。壊せる壁については、壁の近くで爆弾を爆発させると、破壊することができます。壁を壊して通路を作ったり、壁の向こうにいる仲間を助けたり、壁の向こうの敵を攻撃したりといった使い方をします。

「ボンバーマン」でも壁を壊すことができます。こちらも壁の破壊には爆弾を使います。また、壁ではありませんが、「ディグダグ」では土を掘って通路を作ります。

**Fig. 3-90** 壁に道をふさがれている

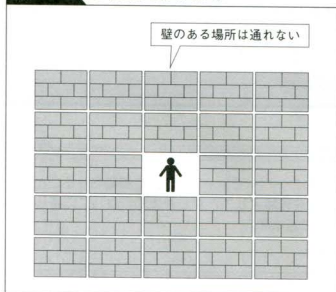
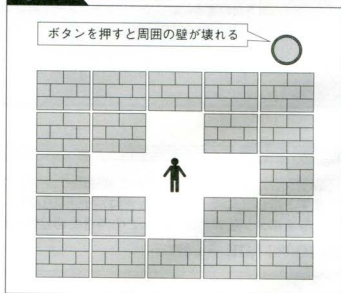



**Fig. 3-91** 周囲の壁を壊す

## アルゴリズム Algorithm

壁を壊す方法はゲームによっていろいろですが、例えばキャラクターがボタンを押したときに、キャラクターから一定距離内にある壁を壊すようにするとよいでしょう（Fig. 3-93）。図では、上下左右の1マスずつの範囲が壁を壊れています。

爆弾を使う場合には、キャラクターからではなく、爆発地点から一定距離内にある壁を壊すようにします。

Fig. 3-92 壁を壊して通路を作りながら進む

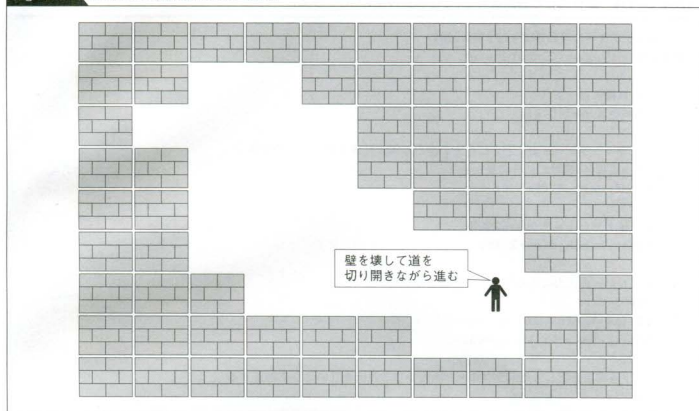
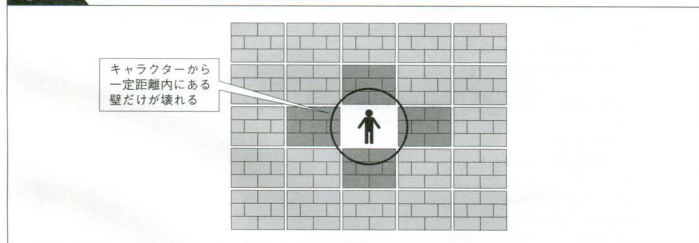



Fig. 3-93 キャラクターから一定距離内にある壁を壊す

## プログラム Program

List 3-14は壁を壊して通路を作るプログラムです。このサンプルでは壊した壁を即座に消去していますが、壁が壊れる様子をアニメーションで表示すれば、壁を壊すアクションがより楽しくなるでしょう。

**List 3-14** 壁を壊して通路を作る（CBreakWallManクラス）

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // キャラクターが歩くときに壁との当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float walk_diff=1.0f;

    // 壊す壁とキャラクターの距離
    float break_dist=2.1f;

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // レバーの入力に応じて上下に移動する
    VY=0;
    if (is->Up) VY=-speed;
    if (is->Down) VY=speed;

    // Y座標を更新し、キャラクターが画面からはみ出さないように補正する
    Y+=VY;
    if (Y<0) Y=0;
    if (Y>MAX_Y-1) Y=MAX_Y-1;

    // 壁との当たり判定処理
    // 壁は通り抜けられないので、
    // 壁に接触したら、X座標とY座標の更新をキャンセルする
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==1 &&
            abs(mover->X-X)<walk_diff &&
            abs(mover->Y-Y)<walk_diff
        ) {
            X-=VX;
            Y-=VY;
            break;
        }
    }
```

```
    // 壁を壊す処理
    // ボタンを押したら、キャラクターから一定距離内にある壁を壊す
    // すべての壁についてキャラクターとの距離を調べ、
    // 一定距離内ならば壁を消去する
    if (is->Button[0]) {
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (mover->Type==1) {
                float dx=mover->X-X;
                float dy=mover->Y-Y;
                if (sqrt(dx*dx+dy*dy)<break_dist) {
                    i.Remove();
                }
            }
        }
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「BREAKING WALL」は壁を壊して通路を作るアクションのサンプルです。レバーでキャラクターを上下左右に移動させることができます。ボタンを押すとキャラクターに隣接する壁を消します。壁が消えた部分はキャラクターが移動可能になります。

**BREAKING WALL** → p. 395

## まとめ Stage03

本章では、アクションゲームのステージを構成するさまざまな「仕掛け」について解説しました。多彩な仕掛けを盛り込んだゲームを作るのもよいのですが、1つの仕掛けにとことんこだわって、奥深く掘り下げたゲームも魅力があります。また、既存の仕掛けを研究しつつ、新しい仕掛けを考え出すのも楽しいものです。

というわけで、「魅力的な仕掛けをステージに組み込んで、プレイヤーをうならせよう！」というのが本章のまとめです。

アクションゲームのステージは、山あり谷ありの起伏に富んだ地形が一般的です。岩や氷といった障害物が配置されていることもあります。キャラクターは危険な地形や障害物を回避するだけではなく、ときにはこれらを逆に利用して敵を攻撃するなど、多彩なアクションを見せます。

# 地形利用

## Land Features

ActionGame Algorithm Maniax

# Stage 04

## 壁や天井に張り付く

キャラクターが壁や天井に張り付くアクションです。壁に張り付いて上下に移動したり、天井に張り付いて左右に移動したりすることもできます。また、壁からジャンプして、ほかの壁や天井に張り付くことも可能です。

最初に、キャラクターが壁に張り付く状況を見てみましょう。壁に張り付くには、まず壁に近づく必要があります (Fig. 4-1)。壁の近くから、壁に向かってジャンプすると、壁に張り付くことができます (Fig. 4-2)。

壁に張り付いた状態でジャンプボタンを押すと、キャラクターがジャンプして壁から離れます。上手にジャンプすれば、壁から高いところにある床へ飛び移ることもできます (Fig. 4-3)。

**Fig. 4-1** キャラクターと壁

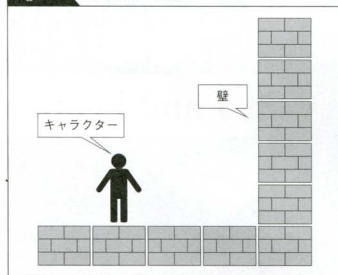



**Fig. 4-2** 壁に張り付く

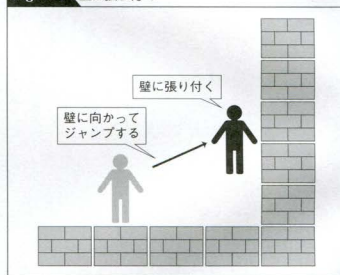



**Fig. 4-3** 壁から離れる

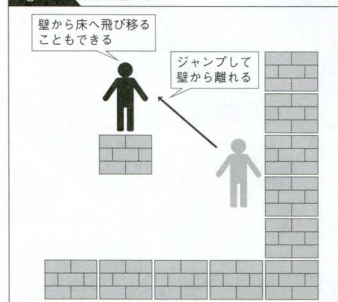



**Fig. 4-4** 縦穴を登る

壁と壁が近いときには、もっと面白い動きができます(Fig. 4-4)。壁から壁へのジャンプを繰り返して、縦穴を登っていくことが可能です。

次に、キャラクターが天井に張り付く状況について見てみます(Fig. 4-5)。天井に向かってジャンプすると、天井に張り付くことができます。

天井から離れるには、壁の場合と同じように、ジャンプボタンを押します(Fig. 4-6)。キャラクターは飛び降りて、天井から離れることができます。

壁や天井に張り付くことができるのは、主人公が忍者のゲームが多いようです。壁や天井に張り付いて移動するというアクションが、いかにも忍者らしいからでしょう。例えば「忍者龍剣伝」や「ストライダー飛竜」は、どちらも忍者が主人公のゲームです。壁や天井に張り付いたり、そのまま上下左右に移動したり、別の壁や天井に飛び移ったりすることができます。

忍者以外を主人公にしたゲームでは、「ちゃっくんぽっぷ」でも天井に張り付くことができます。このゲームではキャラクターが逆さまになって天井に張り付き、天井を左右に移動したり、天井にある起伏を乗り越えたりすることが可能です。

壁や天井に張り付くゲームを面白くするには、ステージの構成が重要です。張り付くアクションを生かせるように、張り付きやすい壁や天井を用意したり、飛び移りやすいように壁や天井の間隔を設定したりといった調整をする必要があります。こういったゲームのステージは、アクションゲームのなかでも特に起伏に富んだ地形になります。

**Fig. 4-5** 天井に張り付く

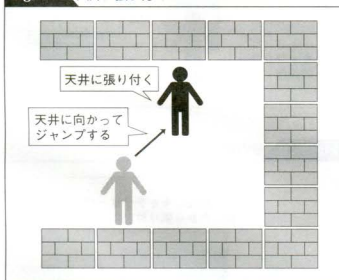



**Fig. 4-6** 天井から離れる

## アルゴリズム Algorithm

壁や天井に張り付くアクションを実現するには、キャラクターが空中にいる状態と、キャラクターが何かに張り付いている状態を区別します。ここではJumpというフラグを用意して、キャラクターが空中にいるとき、つまりジャンプまたは落下しているときにはフラグをtrueにすることにします(Fig. 4-7)。壁や天井に張り付いていたり、床に乗ったりしているときには、フラグをfalseにします。

キャラクターが天井に張り付いたときには、フラグをfalseにします（Fig. 4-8）。天井に張り付く条件は次の2つです。

・キャラクターが天井から一定範囲内にいる
・キャラクターの速度が上向きである

キャラクターの速度に関する条件があるのは、キャラクターが天井に飛び付いたときと、天井から離れて落下するときを区別するためです。天井に飛び付いたときだけ張り付かせるために、キャラクターが上昇中かどうか、つまり速度が上向きかどうかを調べます。

同様に、壁と床についても判定処理を行い、壁に張り付いたかどうか、あるいは床に飛び乗ったかどうかを調べます。キャラクターのX方向の速度をVX、Y方向の速度をVYとすると、天井と床に関する条件はFig. 4-9のように、壁に関する条件はFig. 4-10のようになります。キャラクターが壁・天井・床に近づいてくる方向に、それぞれ当たり判定が広がっていることがポイントです。

**Fig. 4-7** キャラクターの状態を表すフラグ

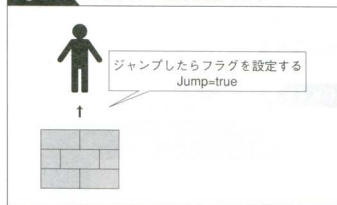



**Fig. 4-8** 天井に張り付く

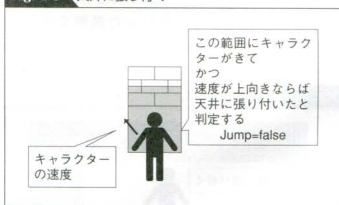



**Fig. 4-9** 天井と床の判定

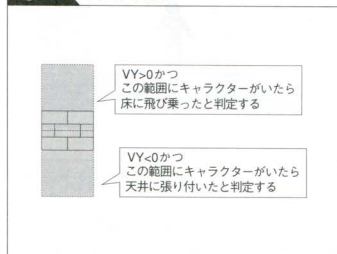



**Fig. 4-10** 壁の判定

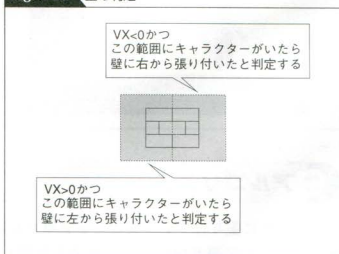



Fig. 4-11　壁に張り付いたまま移動する

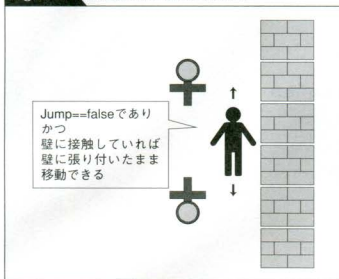



Fig. 4-12　天井に張り付いたまま移動する

フラグがfalse、つまり何かに張り付いている状態のキャラクターは、上下または左右に移動することができます。キャラクターが壁に接触していれば、壁に張り付いたまま上下に移動することができます（Fig. 4-11）。天井に接触しているときと、床に乗っているときには、左右に移動することができます（Fig. 4-12）。

## プログラム　Program

List 4-1は壁や天井に張り付くアクションのプログラムです。壁や天井に張り付いて移動したり、壁から壁へ、あるいは壁から天井へと飛び移ったりすることができます。壁・天井・床について、それぞれ当たり判定処理を行うことがポイントです。

**List 4-1**　壁や天井に張り付く（CHangOnWallManクラス）

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // X方向の移動スピード
    float speed=0.2f;

    // ジャンプの初速度
    float jump_speed=-0.3f;

    // ジャンプ中の加速度
    float jump_accel=0.01f;

    // 壁・天井・床との当たり判定処理を行うための定数
    // X座標の差分の最大値、Y座標の差分の最小値と最大値
```

## List 4-1

```
float max_diff=0.6f;
float min_dist=0.0f;
float max_dist=1.3f;

// 壁・天井・床に接触したときに、
// キャラクターを壁・天井・床から適切な距離に戻すための定数
float adjust_dist=1.0f;

// 上の天井・左の壁・右の壁・下の床に、
// キャラクターが接触しているかどうかのフラグ
bool hit_up=false;
bool hit_left=false;
bool hit_right=false;
bool hit_down=false;

// 壁・天井・床にキャラクターが接触したかどうかの判定処理
for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
    CMover* mover=(CMover*)i.Next();
    if (mover->Type==1) {

        // 天井に張り付いたかどうかの判定処理
        if (
            abs(X-mover->X)<max_diff &&
            Y-mover->Y>=min_dist &&
            Y-mover->Y<max_dist
        ) {
            // 天井に接触したフラグをtrueにする
            hit_up=true;

            // 速度が上向きならば天井に張り付く
            // キャラクターと天井の距離を調整し、
            // 空中にいるフラグをfalseにする
            // Jumpは空中にいるかどうかを表すフラグ
            if (VY<0) {
                Y=mover->Y+adjust_dist;
                Jump=false;
            }
        }

        // 床に飛び乗ったかどうかの判定処理
        if (
            abs(X-mover->X)<max_diff &&
            mover->Y-Y>=min_dist &&
            mover->Y-Y<max_dist
        ) {
            // 床に接触したフラグをtrueにする
            hit_down=true;

            // 速度が下向きならば床に飛び乗る
            // キャラクターと床の距離を調整し、
```

```
            // 空中にいるフラグをfalseにする
            if (VY>0) {
                Y=mover->Y-adjust_dist;
                Jump=false;
            }
        }

        // 左の壁に張り付いたかどうかの判定処理
        if (
            abs(Y-mover->Y)<max_diff &&
            X-mover->X>=min_dist &&
            X-mover->X<max_dist
        ) {
            // 左の壁に接触したフラグをtrueにする
            hit_left=true;

            // 速度が左向きならば壁に張り付く
            // キャラクターと壁の距離を調整し、
            // 空中にいるフラグをfalseにする
            if (VX<0) {
                X=mover->X+adjust_dist;
                Jump=false;
            }
        }

        // 右の壁に張り付いたかどうかの判定処理
        if (
            abs(Y-mover->Y)<max_diff &&
            mover->X-X>=min_dist &&
            mover->X-X<max_dist
        ) {
            // 右の壁に接触したフラグをtrueにする
            hit_right=true;

            // 速度が右向きならば壁に張り付く
            // キャラクターと壁の距離を調整し、
            // 空中にいるフラグをfalseにする
            if (VX>0) {
                X=mover->X-adjust_dist;
                Jump=false;
            }
        }
    }
}

// キャラクターが空中にいるときの処理
// Y方向の速度に加速度を加える
// また、Y方向の速度が大きくなりすぎないように補正する
// 速度が大きくなりすぎると、
// 当たり判定をすりぬけることがあるため
```

## List 4-1

```
if (Jump) {
    VY+=jump_accel;
    if (VY>-jump_speed) VY=-jump_speed;
} else

// キャラクターが空中にいないときの処理
{
    // 天井に張り付いているか、床に乗っているときの処理
    VX=VY=0;
    if (hit_up || hit_down) {

        // レバーの入力に応じて左右に移動する
        if (is->Left && !hit_left) VX=-speed;
        if (is->Right && !hit_right) VX=speed;

        // ジャンプボタンを押したときの処理
        // ボタンを押して、かつ左右に壁がなければ、
        // キャラクターをジャンプさせる
        // 天井にいるときにはY方向の速度を0にして落下させ、
        // 床にいるときには上向きの初速度を設定してジャンプさせる
        if (
            !PrevButton && is->Button[0] &&
            !hit_left && !hit_right
        ) {
            VY=hit_up?0:jump_speed;
            Jump=true;
        }
    }

    // 左右の壁に張り付いているときの処理
    if (hit_left || hit_right) {

        // レバーの入力に応じて上下に移動する
        if (is->Up && !hit_up) VY=-speed;
        if (is->Down && !hit_down) VY=speed;

        // ジャンプボタンを押したときの処理
        // ボタンを押して、かつ上に天井がなければ、
        // キャラクターをジャンプさせる
        // X方向の速度は壁から離れる向きに設定し、
        // Y方向には上向きの初速度を設定する
        if (
            !PrevButton && is->Button[0] &&
            !hit_up
        ) {
            VX=hit_left?speed:-speed;
            VY=jump_speed;
            Jump=true;
        }
    }
```

```
        // 壁・天井・床のいずれにも接触していないときの処理
        // キャラクターが空中にいるフラグをtrueにする
        if (!hit_up && !hit_down && !hit_left && !hit_right) {
            Jump=true;
        }
    }

    // X座標とY座標の更新
    X+=VX;
    Y+=VY;

    // ジャンプボタンを押した瞬間を判別するため、
    // ボタンの状態を保存する
    PrevButton=is->Button[0];

    // X方向の速度に応じて、キャラクターを傾けて表示する
    // キャラクターが壁に張り付いているときには、
    // 壁に頭を近づけるように傾ける
    if (Jump || hit_up || hit_down) {
        Angle=VX/speed*0.1f;
    } else {
        Angle=hit_left?-0.1f:0.1f;
    }

    return true;
}
```

SAMPLE

「HANGING ON WALL」は壁や天井への張り付きのアクションのサンプルです。レバーでキャラクターを動かし、ボタンでジャンプします。ジャンプした先に壁や天井があると、キャラクターはそこに張り付きます。張り付いているときは、壁や天井に沿って移動することができます。

**HANGING ON WALL** → p. 395

# 岩落とし

岩を落として道を作ったり、敵をつぶしたりするアクションです。地中に埋まっている岩の場合には、岩の下にある土を掘れば、岩を落とすことができます。一方、岩を押して崖から落とすゲームもあります。

ここでは地中に埋まっている岩について考えましょう (Fig. 4-13)。キャラクターも地中にいます。キャラクターを移動させると、土を掘って通路を作ることができます。このときに岩の下の土を掘ると、岩は支えがなくなって、グラグラと揺れ始めます (Fig. 4-14)。

支えがなくなった岩は、やがて落下します (Fig. 4-15)。落ちた岩は、下に土や別の岩がある場所まで落下を続けます。落下を終えた岩が壊れてしまうか、その場に残るかは、ゲームによって違います。

Fig. 4-13 地中に埋まっている岩

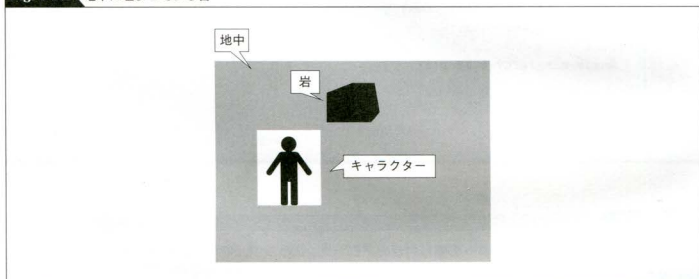



Fig. 4-14 岩の下を掘る

Fig. 4-15 岩が落下する

岩落としを採用したゲームには「ディグダグ」があります。このゲームでは、地中に埋まっている岩の下を掘ると、岩を落とすことができます。落下する岩の下にうまく敵を誘い込めば、敵をつぶして倒すことができます。敵をポンプで膨らませて動けなくさせておいたうえで岩を落とす、といった戦術も楽しめます。

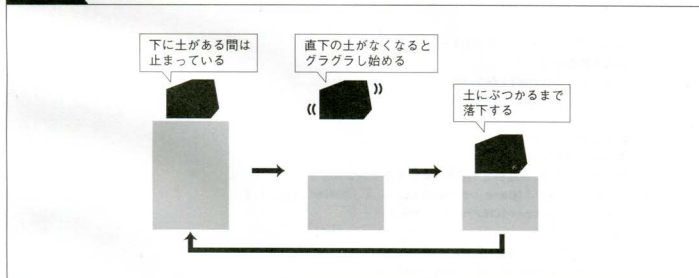
「バーガータイム」にも、岩落としに似たアクションがあります。このゲームではハンバーガーの材料を落とします。キャラクターが材料の上を通過すると、材料を落とすことができます。材料を落として下にいる敵をつぶしたり、材料の上に敵がいるときに敵ごと落としたり、といった攻撃が可能です。

## アルゴリズム Algorithm

岩落としを実現するには、岩の状態を管理することがポイントです(Fig. 4-16)。岩には次の3つの状態があります。

- 下に土があり、止まっている状態
- 下の土がなくなって、グラグラ揺れている状態
- 落下している状態

下の土がなくなると、岩は少しの間グラグラと揺れたあとに落下します。岩は土や別の岩にぶつかるまで落下し、ぶつかると止まって、最初の状態に戻ります。

**Fig. 4-16** 岩の状態

## プログラム Program

List 4-2は岩落としのプログラムです。このサンプルでは、キャラクターが地中を動くと、土を掘ることができます。この効果は、土とキャラクターが接触したかどうかを調べ、接触した土を消去することによって実現しています。

**List 4-2** 岩落とし(CDropRockManクラス、CDropRockクラス、CDropRockClayクラス)

```
// キャラクターの移動処理を行うMove関数
bool CDropRockMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 岩との当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=1.0f;

    // レバーの入力に応じて左右上下に移動する
    VX=VY=0;
    if (is->Left) VX=-speed; else
    if (is->Right) VX=speed; else
    if (is->Up) VY=-speed; else
    if (is->Down) VY=speed;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // Y座標を更新し、キャラクターが画面からはみ出さないように補正する
    Y+=VY;
    if (Y<0) Y=0;
    if (Y>MAX_Y-1) Y=MAX_Y-1;

    // キャラクターが岩や土のサイズと同じ格子上を動くように、
    // 座標を調整する
    if (VX!=0) Y=(int)(Y+0.5f);
    if (VY!=0) X=(int)(X+0.5f);

    // 岩との当たり判定処理
    // 岩は通り抜けられないので、
    // 岩に接触したら、X座標とY座標の更新をキャンセルする
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==2 &&
            abs(X-mover->X)<max_dist &&
            abs(Y-mover->Y)<max_dist
        ) {
            X-=VX;
            Y-=VY;
            break;
        }
    }
```

```
    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}

// 岩の移動処理を行うMove関数
bool CDropRock::Move(const CInputState* is) {

    // 落下スピード
    float speed=0.2f;

    // 土との当たり判定処理を行うための定数
    // Y座標の差分の最大値
    float max_dist=1.2f;

    // 岩が落下するかどうかを調べる
    // 最初は落下しているかどうかのフラグをtrueにしておく
    bool drop=true;

    // 画面の下端に達したら、落下のフラグをfalseにする
    if (Y>=MAX_Y-1) {
        drop=false;
    } else

    // 岩の下に土があるかどうかを調べる
    // 土があるときには、落下中のフラグをfalseにする
    {
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (
                X==mover->X &&
                mover->Y-Y>0 &&
                mover->Y-Y<=max_dist
            ) {
                drop=false;
                break;
            }
        }
    }

    // 状態に応じて分岐する
    // Stateは岩の状態を表す変数
    switch (State) {

        // 静止状態
        // 下の土がなくなったら、残り時間を設定して、グラグラ状態に移行する
        case 0:
            if (drop) {
                Time=60;
```

**List 4-2**

```
                State=1;
            }
            break;

        // グラグラ状態
        // 残り時間があるかぎり、グラグラと揺れる
        // 残り時間がなくなったら、落下状態に移行する
        case 1:
            Time--;
            if (Time==0) State=2;
            break;

        // 落下状態
        // 下に土がないかぎり、落下を続ける
        // 土に当たったり、画面下端に達したりしたら、
        // 静止状態に移行する
        case 2:
            if (!drop) {
                State=0;
            } else {
                Y+=speed;
            }
            break;
    }

    // グラグラ状態では、画像を傾けて揺れている様子を表現する
    // それ以外の状態では、画像を傾けずに表示する
    if (State==1) Angle=sin(Time*0.5f)*0.05f; else Angle=0;

    return true;
}

// 土の移動処理を行うMove関数
// キャラクターとの当たり判定処理を行い、
// キャラクターが接触したら、土を消去する
bool CDropRockClay::Move(const CInputState* is) {
    float max_dist=0.5f;
    return
        abs(X-Man->X)>=max_dist ||
        abs(Y-Man->Y)>=max_dist;
}
```

**SAMPLE**

「DROPPING ROCK」は岩落としのサンプルです。レバーでキャラクターを上下左右に移動させることができます。キャラクターは移動しながら地中を掘り進んでいきます。岩の下の土を掘ると、岩が落下します。

**DROPPING ROCK** → p. 395

# ものを押して動かす

ブロックなどのものをキャラクターが押して動かすアクションです。進路をじゃまするものを脇へ動かしたり、積み上げて通路を作ったり、落として敵をつぶしたりと、ものを押すアクションには実にさまざまなバリエーションがあります。

ここではブロックを押すことを考えましょう(Fig. 4-17)。ブロックを押すには、キャラクターをブロックに押しつけるように移動させます(Fig. 4-18)。すると、押されたブロックがキャラクターと同じ方向に移動します。

押したブロックの先に障害物があるとき、例えば別のブロックがあるような場合には、ブロックを押すことができません(Fig. 4-19)。多くのゲームでは、ブロックを1つずつしか押せません。ブロックの先にまたブロックがある場合には、まず先にあるブロックを動かして、進路を確保しておく必要があります。

ものを押して動かすゲームは非常に数多くあります。例えば「フラッピー」は、青と赤の2種類のブロックを押して、青いブロックをゴールまで運ぶゲームです。ブロックを重ねて通路を作るようなパズル要素と、ブロックを落として敵をつぶすようなアクション要素が、ほどよくミックスされています。

「パックランド」では、消火栓や木などを押すことができます。特定の場所にある物体を進行方向とは逆に押すと、アイテムが出現したり、先の面にワープできたりします。これは物体を押すアクションが隠し要素として使われている例です。

また「フェアリーランドストーリー」では、敵を攻撃してお菓子に変えたあとに、そのお菓子を押すことができます。お菓子を敵の上に落とすと、敵をつぶして倒すことができます。

**Fig. 4-17** キャラクターとブロック

キャラクター

ブロック

**Fig. 4-18** ブロックを押す

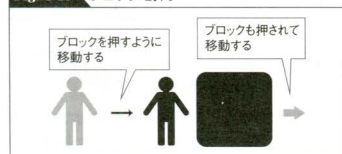



**Fig. 4-19** 移動先に障害物がある場合

## アルゴリズム Algorithm

ブロックを例に、ものを押して動かすアクションを実現する方法を考えてみましょう。

まず、キャラクターがブロックに接触しているかどうかを調べます(Fig. 4-20)。キャラクターがブロックから一定範囲内にいるときは、接触していると判定します。

キャラクターが接触していたら、ブロックをキャラクターと同じ移動方向へ動かします(Fig. 4-21)。例えばキャラクターの速度が(VX, VY)ならば、ブロックを速度(VX, VY)で動かします。これで、キャラクターがブロックを押したように見えます。

ブロックの先に障害物がある場合には、ブロックを動かすことはできません(Fig. 4-22)。そこで、移動後のブロックが、障害物に接触するかどうかを調べます。接触する場合には、ブロックの移動をキャンセルします。ブロックを押しているキャラクターも動くことができないので、キャラクターの移動もキャンセルします。

**Fig. 4-20** ブロックを押しているかどうかの判定処理

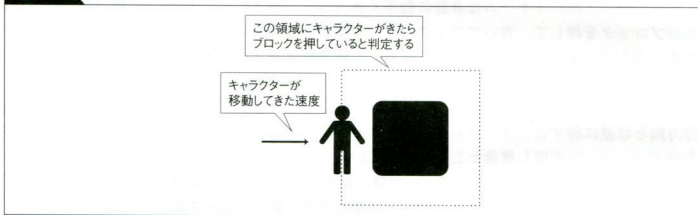



**Fig. 4-21** ブロックを動かす

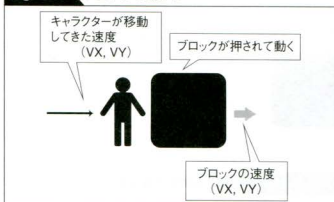



**Fig. 4-22** ブロックが動かせない場合

## プログラム Program

List 4-3はものを押して動かすアクションのプログラムです。ブロックを押す処理では、まずキャラクターがブロックに接触したかどうかを調べます。次に、押されたブロックがほかの

ブロックに接触するかどうかを調べます。接触しないときにはキャラクターとブロックを移動させます。接触するときにはどちらも移動させません。

**List 4-3** ものを押して動かす(CPushObjectManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // ブロックとの当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    // 値を大きくしすぎると、ブロックとブロックの間に
    // キャラクターが位置したときにブロックを押せなくなる
    float max_dist=0.9f;

    // レバーの入力に応じて上下左右に移動する
    VX=VY=0;
    if (is->Left) VX=-speed; else
    if (is->Right) VX=speed; else
    if (is->Up) VY=-speed; else
    if (is->Down) VY=speed;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // Y座標を更新し、キャラクターが画面からはみ出さないように補正する
    Y+=VY;
    if (Y<0) Y=0;
    if (Y>MAX_Y-1) Y=MAX_Y-1;

    // ブロックにキャラクターが接触したかどうかの判定処理
    int count=0;
    CMover* object=NULL;
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==1 &&
            abs(X-mover->X)<max_dist &&
            abs(Y-mover->Y)<max_dist
        ) {
            // 接触したブロックを記録しておく
            object=mover;

            // 同時に2つ以上のブロックに接触している場合には、押すことがでいきないので、
```

**List 4-3**

```
			// キャラクターの移動をキャンセルする
			count++;
			if (count>1) {
				X-=VX;
				Y-=VY;
				break;
			}
		}
	}

	// 1つのブロックだけに接触している場合
	// ブロックを押す処理
	if (count==1) {

		// 押すブロックの先に障害物がないかどうかを調べる
		// 押すブロックの移動先に別のブロックがあったり、
		// 移動先が画面外だったりしたら、
		// 移動の可否を表すフラグ(movable)をfalseにする
		bool movable=true;
		float x=object->X+VX;
		float y=object->Y+VY;
		for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
			CMover* mover=(CMover*)i.Next();
			if (
				x<0 || x>MAX_X-1 ||
				y<0 || y>MAX_Y-1 ||
				mover!=object &&
				mover->Type==1 &&
				abs(x-mover->X)<max_dist &&
				abs(y-mover->Y)<max_dist
			) {
				movable=false;
				break;
			}
		}

		// 移動できる場合には、ブロックを移動させる
		if (movable) {
			object->X=x;
			object->Y=y;
		} else

		// 移動できない場合には、ブロックは移動させない
		// キャラクターの移動もキャンセルする
		{
			X-=VX;
			Y-=VY;
		}
	}
```

```
    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「PUSHING OBJECT」はものを押すアクションのサンプルです。レバーでキャラクターを上下左右に移動することができます。またキャラクターでブロックを押すことができます。ただし移動先に別のブロックがある場合は押すことはできません。

**PUSHING OBJECT** → p. 395

## 氷を押す

氷を押してすべらせるアクションです。ものを押して動かすアクションに似ていますが、氷はよくすべるので、何か別のものにぶつかって止まるまで勝手に進んでいきます。氷を動かして通路を作ったり、氷で敵をつぶしたりすることができます。

氷を押すアクションは、まず氷に近づくことから始まります(Fig. 4-23)。キャラクターが氷に接触した状態でボタンを押すと、氷を押すことができます(Fig. 4-24)。

押された氷はすべり始めます(Fig. 4-25)。氷はまっすぐすべっていき、別の氷などの障害物にぶつかるまですべり続けます(Fig. 4-26)。

氷を押すアクションを採用したゲームには、例えば「ペンゴ」があります。このゲームでは、氷を押してすべらせることによって、通路を作ったり、敵を倒したりできます。氷を別の氷に押しつけて、壊すことも可能です。

また、氷と同じようにすべらせることができる特別なブロックが、ステージに3つあります。3つのブロックを一列に並べると、ボーナス得点が入るようになっています。ちょっとしたパズルとして楽しめる工夫です。

**Fig. 4-23** 氷に近づく

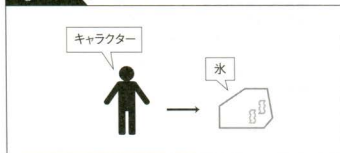



**Fig. 4-24** 氷の近くでボタンを押す

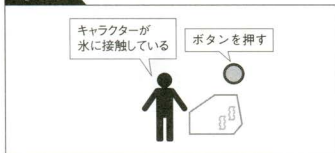



**Fig. 4-25** 氷がすべり始める

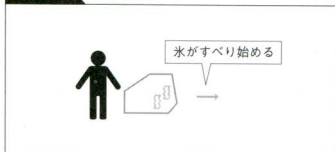
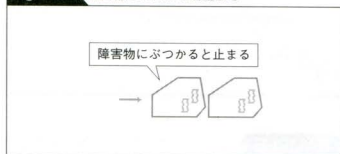



**Fig. 4-26** 障害物にぶつかると止まる

## アルゴリズム Algorithm

氷を押すアクションを実現するには、まずキャラクターが氷を押せる位置にいるかどうかを判定します(Fig. 4-27)。キャラクターが移動してきた方向を考慮して、キャラクターの前方に氷があるかどうかを調べます。例えばキャラクターが左から右へ移動してきたら、キャラクターの少し右を調べて、そこに氷があるかどうかを判定します。

キャラクターの前方に氷がある状態でボタンを押したら、氷をすべらせます。氷をすべらせる方向は、キャラクターが移動してきた方向と同じです。例えばキャラクターが左から右へ移動してきたら、氷を右にすべらせます。すべり出した氷は、特にレバーなどで操作しなくても自動的に進んでいきます。そして、別の氷などの障害物に接触すると、すべるのを中断します(Fig. 4-28)。

**Fig. 4-27** 氷に接触しているかどうかの判定処理

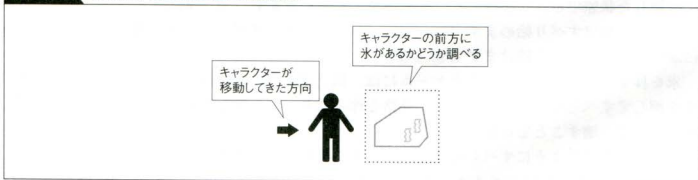
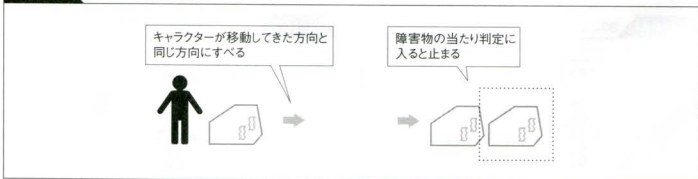



**Fig. 4-28** 氷のすべり始めと停止

## プログラム Program

List 4-4は氷を押すアクションのプログラムです。このプログラムでは、変数DirXとDirYにキャラクターが移動した方向を保存しておきます。保存した値は、キャラクターが押す氷を選ぶときと、氷をすべらせる方向を決めるときに使います。

**List 4-4** 氷を押す(CPushIceクラス、CPushIceManクラス)

```
// 氷の移動処理を行うMove関数
bool CPushIce::Move(const CInputState* is) {

    // ほかの氷との当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=0.9f;

    // X座標を更新し、氷が画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // Y座標を更新し、氷が画面からはみ出さないように補正する
    Y+=VY;
    if (Y<0) Y=0;
    if (Y>MAX_Y-1) Y=MAX_Y-1;

    // ほかの氷との当たり判定処理
    // ほかの氷から一定距離内に入ったかどうかを調べる
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover!=this &&
            mover->Type==1 &&
            abs(X-mover->X)<max_dist &&
            abs(Y-mover->Y)<max_dist
        ) {
            // 移動をキャンセルし、速度を0にする
            X-=VX;
            Y-=VY;
            VX=VY=0;

            // 氷のサイズと同じ格子上に並ぶように、氷の座標を調整する
            X=(int)(X+0.5f);
            Y=(int)(Y+0.5f);

            break;
        }
    }
```

**List 4-4**

```
    return true;
}

// キャラクターの移動処理を行うMove関数
bool CPushIceMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 氷の移動スピード
    float ice_speed=0.4f;

    // 氷との当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=0.9f;

    // レバーの入力に応じて左右上下に移動する
    // 氷を押したときのために、
    // キャラクターが移動した方向を保存しておく
    // VXとVYはキャラクターの速度を表す変数
    // DirXとDirYはキャラクターの移動方向を表す変数
    VX=VY=0;
    if (is->Left) {
        DirX=-1;
        DirY=0;
        VX=-speed;
    } else
    if (is->Right) {
        DirX=1;
        DirY=0;
        VX=speed;
    } else
    if (is->Up) {
        DirX=0;
        DirY=-1;
        VY=-speed;
    } else
    if (is->Down) {
        DirX=0;
        DirY=1;
        VY=speed;
    }

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // Y座標を更新し、画面からはみ出さないように補正する
    Y+=VY;
```

```
	if (Y<0) Y=0;
	if (Y>MAX_Y-1) Y=MAX_Y-1;

	// 氷との当たり判定処理
	// 氷は通り抜けられないので、氷に接触したら、X座標とY座標の更新をキャンセルする
	for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
		CMover* mover=(CMover*)i.Next();
		if (
			mover->Type==1 &&
			abs(X-mover->X)<max_dist &&
			abs(Y-mover->Y)<max_dist
		) {
			X-=VX;
			Y-=VY;
			break;
		}
	}

	// ボタンを押したときに氷を押す処理
	// キャラクターが移動してきた方向を考慮して、キャラクターの前方に氷があるかどうかを調べる
	// ここでは前方の座標(X+DirX, Y+DirY)が、氷から一定範囲内にあるかどうかを調べている
	if (!PrevButton && is->Button[0]) {
		for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
			CMover* mover=(CMover*)i.Next();
			if (
				mover->Type==1 &&
				abs(X+DirX-mover->X)<max_dist &&
				abs(Y+DirY-mover->Y)<max_dist
			) {
				// 前方に氷が見つかったら、氷に速度を与えてすべらせる
				// 氷をすべらせる方向は、キャラクターが移動してきた方向にする
				CPushIce* ice=(CPushIce*)mover;
				ice->VX=DirX*ice_speed;
				ice->VY=DirY*ice_speed;
				break;
			}
		}
	}

	// ボタンを押した瞬間を判定するために、現在のボタンの状態を保存しておく
	PrevButton=is->Button[0];

	// X方向の速度に応じて、キャラクターを傾けて表示する
	Angle=VX/speed*0.1f;

	return true;
}
```

SAMPLE

「PUSHING ICE」は氷を押すアクションのサンプルです。レバーでキャラクターを上下左右に移動することができます。氷に近づいてボタンを押すと、氷を押すことができます。押された氷はすべり出し、別の氷にぶつかるか画面の端に達するまですべり続けます。

**PUSHING ICE** → p. 395

# 自動穴

キャラクターが穴を掘るアクションです。ここで掘った穴は、時間が経つと自動的に埋まります。掘った穴にタイミングよく敵を誘い込むと、敵を埋めて倒すことができます。

まずキャラクターが床に乗っている状況を考えましょう(Fig. 4-29)。ボタンを押すと、床に穴を掘ります(Fig. 4-30)。穴を掘る方法はゲームによって違いますが、例えばボタン0を押すとキャラクターの左側に、ボタン1を押すと右側に穴を掘ることができます。

掘った穴は、時間が経つと自動的に埋まり、床に戻ります(Fig. 4-31)。穴を埋める操作は必要ありません。

穴を通過しようとすると、敵は穴に落ちます(Fig. 4-32)。敵は一定時間が経過すると、穴からはい出てきます(Fig. 4-33)。うまくタイミングを合わせて敵を穴に落とすと、敵が落ちている間に穴が埋まって、敵を倒すことができます(Fig. 4-34)。

キャラクターが穴を通過しようとしたときの動きは、ゲームによって違います。穴を通過できないこともあれば、穴に落ちることもあります。例えば、通常は穴を通過できないが、敵が落ち込んでいる間だけは穴の上を通ることができる、といったルールにしてもよいでしょう(Fig. 4-35)。

自動穴を採用したゲームには「ロードランナー」などがあります。このゲームでは、掘った穴に敵を落としたり、敵を埋めて倒したりといったことができるほか、自分が穴に入ることも可能です。穴に入った状態で、さらに下の床に穴を掘ることもできます。地中深くに埋められ

**Fig. 4-29** 床に乗っているキャラクター

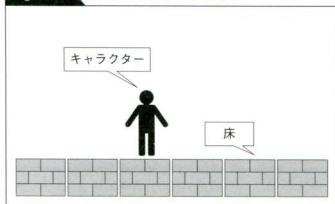



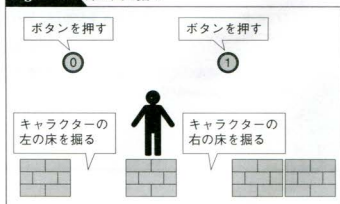
**Fig. 4-30** 床に穴を掘る

た金塊を回収するために、非常に深い穴を掘るテクニックが要求されるステージもあります。穴を利用して敵から逃れるだけではなく、穴や敵を使ったパズル要素が満載で、ステージごとにじっくりと謎解きが楽しめるゲームです。

**Fig. 4-31** 穴は自動的に埋まる

**Fig. 4-32** 敵が穴に落ちる

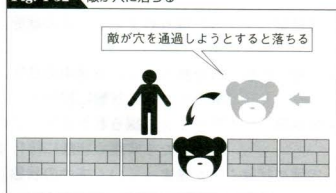
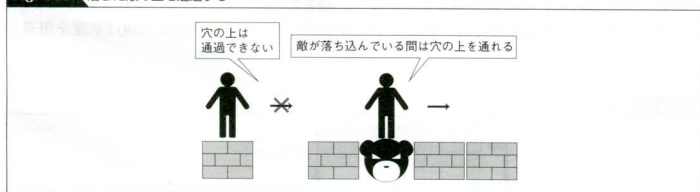



**Fig. 4-33** 敵が穴からはい出てくる

**Fig. 4-34** 敵を穴に埋めて倒す

**Fig. 4-35** 落ちた敵の上を通過する

## アルゴリズム Algorithm

自動穴を実現する際のポイントは、穴の状態を管理することです（Fig. 4-36）。穴には次の4つの状態があります。

- 床状態　　：穴が掘られていない床の状態。キャラクターが穴を掘ったら穴掘り状態に移行する
- 穴掘り状態：穴が掘られている途中の状態。時間とともに穴が深くなっていく。穴が完全に掘られたら穴状態に移行する
- 穴状態　　：穴が完全に掘られた状態。この状態で敵が通過すると穴に落ちる。一定時間が経過したら穴埋め状態に移行する
- 穴埋め状態：穴が埋まっていく途中の状態。時間とともに穴が浅くなっていく。穴が完全に埋まったら床状態に戻る

床状態と穴状態の2つだけでもよいのですが、これでは穴が一瞬で掘られてしまいます。穴は滑らかに掘られた方が見た目によいので、穴掘り状態を設けています。同様に、穴を滑らかに埋めるために、穴埋め状態を設けます。

穴の状態と同様に、敵の状態も管理する必要があります（Fig. 4-37）。敵には次の4つの状態があります。

- 通常状態　：穴に落ちていない状態。床の上を移動し、穴に落ちると穴落ち状態に移行する
- 穴落ち状態：穴に落ちていく状態。穴底まで落ちると待機状態に移行する
- 待機状態　：穴にはまっている状態。一定時間が経過すると穴抜け状態に移行する
- 穴抜け状態：穴から抜け出しつつある状態。穴から完全に抜け出すと通常状態に戻る

敵の状態管理は、穴の状態管理に似ています。通常状態と待機状態だけでもよいのですが、穴に落ちる動作と穴から抜ける動作を滑らかに行うために、穴落ち状態と穴抜け状態を用意しています。

**Fig. 4-36** 穴の状態

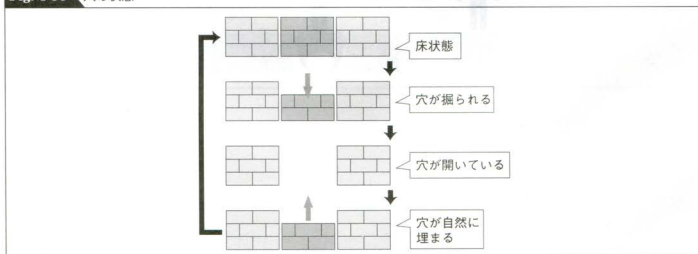
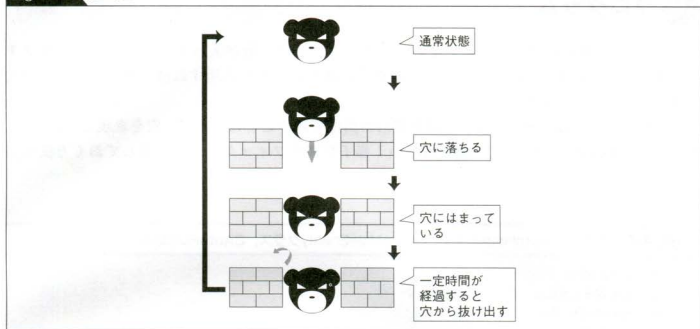



Fig. 4-37 敵の状態

敵が穴に落ちるためには、次の条件を満たす必要があります。

- 敵が穴から一定距離内にいる
- 穴が完全に掘られた状態になっている
- 穴に別の敵が入っていない

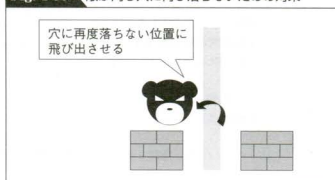
最後の条件は、1つの穴に複数の敵が入らないようにするためのものです。穴ごとに、その穴に敵が落ちているかどうかを記録しておけば、この条件を判定することができます。

敵が穴に落ちたかどうかを判定するには、敵が穴から一定距離内にいるかどうかを調べます(Fig. 4-38)。ここで注意が必要なのは、敵が穴から抜け出したときの処理です。穴から抜け出すときに真上に飛び出すと、再び穴に落ちた判定になってしまいます。

そこで、穴から抜け出すときには、真上よりも少しずらした位置に敵が飛び出すようにします(Fig. 4-39)。敵のX座標を、これから移動する方向に少しずらすとよいでしょう。飛び出したときに、敵が穴から一定距離内にいないようにすれば、再び同じ穴に落ちてしまうことはありません。

Fig. 4-38 敵が穴に落ちたかどうかの判定

Fig. 4-39 敵が同じ穴に再び落ちないための対策

## プログラム　Program

List 4-5は自動穴のプログラムです。このサンプルでは、敵が入っていない穴をキャラクターが通過できないルールにしています。敵が穴に落ちる処理を応用すれば、キャラクターが穴に落ちるようにすることもできます。

このサンプルでは、床の画像の一部を空白の画像で隠すことによって、穴を表現しています。そのほかの手段としては、穴が掘られていく様子をグラフィックとして用意しておく方法もあります。

**List 4-5** 自動穴(CAutoHoleクラス、CAutoHoleEnemyクラス、CAutoHoleManクラス)

```
// 穴を掘るDig関数
// 床に穴を掘るときには、この関数を呼び出す
void CAutoHole::Dig() {

    // 穴を掘るのに要する時間(フレーム数)
    int dig_time=20;

    // 穴が床状態で、かつ敵が埋まっていないときには、
    // 穴掘りの時間を設定して、穴掘り状態に移行する
    if (State==0 && !Enemy) {
        Time=dig_time;
        State=1;
    }
}

// 穴の移動処理を行うMove関数
bool CAutoHole::Move(const CInputState* is) {

    // 穴を掘るのに要する時間(フレーム数)
    int dig_time=20;

    // 穴が埋まり始めるまでの待機時間(フレーム数)
    int wait_time=120;

    // 穴が埋まるのに要する時間(フレーム数)
    int fill_time=20;

    // 状態に応じて分岐する
    // Stateは穴の状態を表す
    switch (State) {

        // 穴掘り状態
        case 1:

            // 残り時間を更新し、時間に応じて深さを更新する
```

```
        // Depthは穴の深さを表す
        Time--;
        Depth=(float)(dig_time-Time)/dig_time;

        // 残り時間が0になったら、
        // 深さと時間を設定し、穴状態に移行する
        if (Time==0) {
            Depth=1;
            Time=wait_time;
            State=2;
        }
        break;

    // 穴状態
    case 2:

        // 残り時間を更新し、時間が0になったら、
        // 時間を設定し、穴埋め状態に移行する
        Time--;
        if (Time==0) {
            Time=fill_time;
            State=3;
        }
        break;

    // 穴埋め状態
    case 3:

        // 残り時間を更新し、時間に応じて深さを更新する
        Time--;
        Depth=(float)Time/fill_time;

        // 残り時間が0になったら、
        // 深さを0に設定し、床状態に戻る
        if (Time==0) {
            Enemy=NULL;
            Depth=0;
            State=0;
        }
        break;
    }

    return true;
}

// 穴の描画処理を行うDraw関数
void CAutoHole::Draw() {

    // 床を描画する
    Texture=Game->Texture[TEX_FLOOR];
```

**List 4-5**

```
    Color=COL_BLACK;
    H=1;
    CMover::Draw();

    // 床の上に重ねて、空白の画像を描画する
    // 穴の深さに応じて画像の大きさを変えることによって、
    // 床が隠れる度合いを調整し、穴が掘られたように見せる
    Texture=Game->Texture[TEX_FILL];
    Color=COL_WHITE;
    float y=Y;
    Y+=(Depth-1)*0.5f;
    H=Depth;
    CMover::Draw();
    Y=y;
}

// 敵の移動処理を行うMove関数
bool CAutoHoleEnemy::Move(const CInputState* is) {

    // 穴に落ちた判定処理を行うための定数
    // X座標の差分の最大値
    float max_dist=0.2f;

    // 穴に落ちるスピード
    float fall_speed=0.1f;

    // 穴に落ちたまま待機している時間（フレーム数）
    int fall_time=60;

    // 状態に応じて分岐する
    // Stateは敵の状態を表す
    switch (State) {

        // 通常状態
        case 0:

            // 左方向に移動し、
            // 画面左端からはみ出したら右端に戻る
            X+=VX;
            if (X<-1) X=MAX_X;

            // 穴に落ちたかどうかの判定処理
            for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
                CMover* mover=(CMover*)i.Next();
                if (
                    mover->Type==1 &&
                    abs(X-mover->X)<max_dist
                ) {
                    Hole=(CAutoHole*)mover;
                    if (Hole->State==2 && !Hole->Enemy) {
```

```
                // 敵が穴に落ちたら、X座標を穴のX座標に合わせてから、
                // 穴落ち状態に移行する
                // また、1つの穴に1つの敵だけを落とすため、
                // 穴のEnemyメンバに、落ちた敵のポインタを記録しておく
                X=Hole->X;
                Hole->Enemy=this;
                State=1;
                break;
            }
        }
    }
    break;

// 穴落ち状態
case 1:

    // Y座標の更新
    Y+=fall_speed;

    // 穴の底まで落ちたら、時間を設定して、待機状態に移行する
    if (Y>=Hole->Y) {
        Time=fall_time;
        State=2;
    }
    break;

// 待機状態
case 2:

    // 残り時間を減らし、
    // 時間が0になったら穴抜け状態に移行する
    Time--;
    if (Time==0) State=3;
    break;

// 穴抜け状態
case 3:

    // Y座標の更新
    Y-=fall_speed;

    // 床の高さまで出たら、
    // X座標とY座標を設定して、通常状態に移行する
    // 出た穴にまたすぐに落ちないために、X座標は穴の真上から少しずらしている
    // また、穴のEnemyメンバにNULLを書き込んで、落ちた敵の記録を消去する
    if (Y<=Hole->Y-1) {
        X=Hole->X-max_dist;
        Y=Hole->Y-1;
        Hole->Enemy=NULL;
```

## List 4-5

```
                State=0;
            }
            break;
    }

    // 穴に埋まったかどうかの判定処理
    // 敵が穴落ち状態・待機状態・穴抜け状態のいずれかのときに、
    // 穴が完全に埋まったら、敵が穴に埋まったと判定する
    // このサンプルでは敵を初期化して再び出現させる
    if (State!=0 && Hole->Depth==0) Init();

    return true;
}

// キャラクターの移動処理を行うMove関数
bool CAutoHoleMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 穴との当たり判定処理を行うための定数
    // X座標の差分の最大値
    float max_dist=0.6f;

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // 穴との当たり判定処理
    // 穴は通過できないので、穴から一定距離内に近づいたら、
    // X座標の更新をキャンセルする
    // ただし、穴に敵が入っているときには通過できる
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==1 &&
            abs(X-mover->X)<max_dist
        ) {
            CAutoHole* hole=(CAutoHole*)mover;
            if (hole->State!=0 && !hole->Enemy) {
                X-=VX;
                break;
            }
        }
```

```
    }

    // ボタン0またはボタン1を押したときに、
    // 左または右に穴を掘る処理
    if (!PrevButton && (is->Button[0] || is->Button[1])) {

        // 穴がキャラクターのちょうど左下または右下になるように、
        // X座標を調整する
        X=(int)(X+0.5f);

        // キャラクターの左下または右下にある床を探し、
        // そこに穴を掘る
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (
                mover->Type==1 &&
                (is->Button[0] && abs(X-1-mover->X)<max_dist) ||
                (is->Button[1] && abs(X+1-mover->X)<max_dist)
            ) {
                CAutoHole* hole=(CAutoHole*)mover;
                hole->Dig();
                break;
            }
        }
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0]||is->Button[1];

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

## SAMPLE

「AUTOMATIC HOLE」は自動穴のサンプルです。レバーでキャラクターを左右に移動させることができます。ボタン0でキャラクターの左側に、ボタン1で右側に穴を掘ることができます。穴は一定時間が経過すると自動的に埋まります。

**AUTOMATIC HOLE** → p. 396

# 手動穴

ボタンの操作で穴を掘り、別のボタンを操作して穴を埋めるアクションです。自動的に穴が埋まることなく、手動で穴を埋め戻さなければならない点が自動穴(→ p. 216)との違いです。掘った穴に敵を誘い込み、敵が抜け出してくる前に素早く埋めると、敵を倒すことができます。

ここでは穴を掘るボタンをボタン0、埋めるボタンをボタン1としましょう。ボタン0を押すと、キャラクターの進行方向に穴を掘ることができます(Fig. 4-40)。

続けてボタン0を押すと、穴を広げることができます(Fig. 4-41)。穴には数段階の大きさがあり、ボタン0を押すたびに穴が広がります。

ボタン1を押すと、掘った穴を埋め戻すことができます(Fig. 4-42)。完全に埋まると穴は消滅して、元の地面に戻ります(Fig. 4-43)。

**Fig. 4-40** 穴を掘る

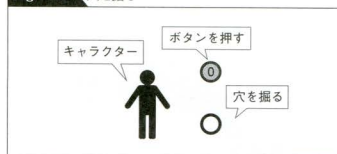



**Fig. 4-41** 穴を広げる

**Fig. 4-42** 穴を埋める

**Fig. 4-43** 穴を完全に埋める

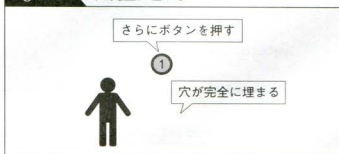



**Fig. 4-44** 敵を穴に誘い込む

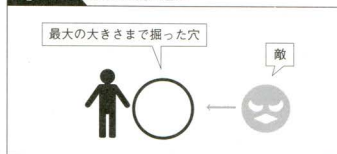
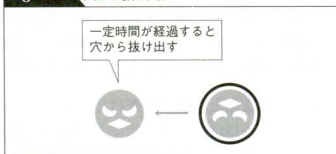



**Fig. 4-45** 敵が穴に落ちる

Fig. 4-46 敵ごと穴を埋める

Fig. 4-47 穴から抜け出す

最大の大きさまで掘った穴には、敵を誘い込んで落とすことができます(Fig. 4-44)。敵は穴の上を通過しようとすると、穴に落ちて、しばらく動けなくなります(Fig. 4-45)。ゲームによっては、穴の大きさが最大でなくても敵を落とすことができます。

敵が穴に落ちたら、動けないでいるうちに素早くボタン1を押して、敵ごと穴を埋めます(Fig. 4-46)。穴を完全に埋めると、敵を倒すことができます。

敵は穴に落ちてから一定時間が経過すると、穴から抜け出します(Fig. 4-47)。敵を倒すには、敵が抜け出してくる前に素早く穴を埋める必要があります。

手動穴を採用したゲームの例としては「平安京エイリアン」があります。このゲームでは、平安京のような碁盤の目に区切られたステージのなかで、道に穴を掘って敵を誘い込みます。敵は穴に落ちるとしばらく動けなくなりますが、一定時間が経過すると、穴を抜け出したうえに穴を埋め戻してしまいます。大きな穴に落ちると敵はしばらく動けなくなりますが、小さな穴はすぐに埋めて出てきてしまいます。

このゲームが面白いのは、ステージの形状を生かして、いろいろな穴掘りの戦術が考案されていることです。例えば、ステージの4つ辻の中心にキャラクターを配置し、上下左右の4方向に穴を掘って敵を待つ「秋葉掘り」や、キャラクターの両側にある辻に2つの穴を掘って敵の侵入を防ぐ「隠居掘り」などのテクニックがあります。技に名前が付いているのも面白いところです。

## アルゴリズム Algorithm

手動穴のポイントは、穴掘りと穴埋めの処理です。穴を掘る場合には、まずボタン0を押したかどうかを判定します(Fig. 4-48)。そして、キャラクターの進行方向に穴があるかどうかを調べます。すでに穴がある場合には、その穴を広げます。穴がない場合には、新しい穴を掘ります。

穴埋めも穴掘りに似ています(Fig. 4-49)。穴を埋める場合には、まずボタン1を押したかどうかを判定します。そして、キャラクターの進行方向に穴があるかどうかを調べます。穴がある場合には、その穴を埋めます。穴がない場合は何もしません。

もう1つのポイントは、敵の動きです(Fig. 4-50)。敵には次の4つの状態があります。

・通常状態　：穴に落ちていない状態。自由に移動し、穴に落ちると穴落ち状態に移行する
・穴落ち状態：穴に落ちていく状態。穴底まで落ちると待機状態に移行する
・待機状態　：穴にはまっている状態。一定時間が経過すると穴抜け状態に移行する
・穴抜け状態：穴から抜け出しつつある状態。穴から完全に抜け出すと通常状態に戻る

敵は通常状態と待機状態だけでもよいのですが、穴に落ちる動作と穴から抜ける動作を滑らかに行うためには、穴落ち状態と穴抜け状態を用意するとよいでしょう。穴落ち状態と穴抜け状態では、敵が穴に出入りする様子をアニメーションなどで表現します。

敵が穴に落ちるための条件は次のように設定しました。これは「自動穴 (→ p. 216)」の場合と同じです。

・敵が穴から一定距離内にいる
・穴が完全に掘られた状態になっている
・穴に別の敵が入っていない

**Fig. 4-48** 穴を掘る処理

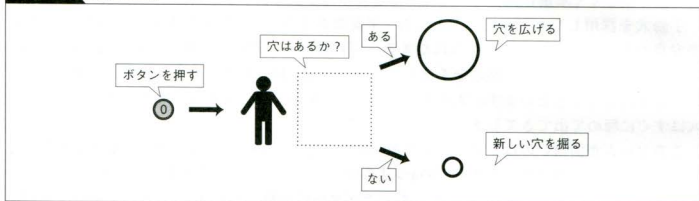



**Fig. 4-49** 穴を埋める処理

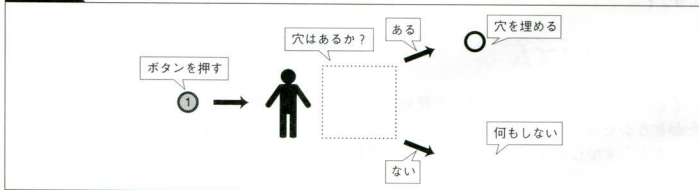
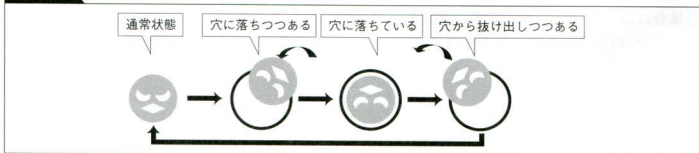



**Fig. 4-50** 敵の行動

「自動穴」の場合と同じく、敵が穴から出るときには、敵を穴から少しずらして飛び出させるとよいでしょう。穴の真上に飛び出させると、穴から出た瞬間に同じ穴に落ちてしまうからです。

## プログラム Program

List 4-6は手動穴のプログラムです。このサンプルでは穴の深さは5段階あります。穴を最大の大きさまで掘るにはボタン0を5回押す必要があり、最大の穴を埋めるにはボタン1を5回押す必要があります。敵が落ちるのは最大の大きさの穴だけで、小さな穴は通過します。

敵が穴に落ちたり穴から抜けたりするときには、敵の画像を回転させて、穴に出入りする様子を表現しました。また、敵が入った穴を埋めるときには、敵の画像を穴の大きさに合わせて縮小表示して、穴に埋まっていく雰囲気を出しました。

このサンプルでは敵が穴から抜ける際に穴を埋めません。敵が穴から抜けたときに、穴の深さを0に戻すようにすれば、穴を埋めることもできます。

**List 4-6** 手動穴(CManualHoleクラス、CManualHoleEnemyクラス、CManualHoleManクラス)

```
// 穴を掘るDig関数
// 地面に穴を掘るときには、この関数を呼び出す
void CManualHole::Dig() {

    // 穴を掘るスピード
    float dig_speed=0.2f;

    // 穴を大きくする
    // Sizeは穴の大きさを表す
    // Sizeの最小値は0(穴がない)、最大値は1(最大の穴がある)
    Size+=dig_speed;

    // 穴が最大の大きさを超えないように補正する
    if (Size>1) Size=1;

    // 穴の大きさに合わせて、穴の画像を表示する
    W=H=Size;
}

// 穴を埋めるFill関数
// 穴を埋めるときには、この関数を呼び出す
void CManualHole::Fill() {

    // 穴を埋めるスピード
    float fill_speed=0.2f;

    // 穴を小さくする。
```

**List 4-6**

```
    Size-=fill_speed;

    // 穴が十分に小さくなったら、
    // 穴のサイズを0にする
    if (Size<0.1f) Size=0;

    // 穴の大きさに合わせて、穴の画像を表示する
    W=H=Size;
}

// 穴の移動処理を行うMove関数
bool CManualHole::Move(const CInputState* is) {

    // 穴のサイズが0になったら、
    // falseを返すことによって、
    // 呼び出し元の関数に穴を消去してもらう
    // ただし、敵が穴に入っている場合には、
    // すぐには消去せず、敵が倒されていなくなるのを待つ
    return Size>0 || Enemy;
}

// 敵の移動処理を行うMove関数
bool CManualHoleEnemy::Move(const CInputState* is) {

    // 穴に落ちた判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=0.4f;

    // 穴に落ちるときの回転スピード
    float fall_vangle=0.05f;

    // 穴に落ちたまま待機している時間(フレーム数)
    int fall_time=120;

    // 状態に応じて分岐する
    // Stateは敵の状態を表す
    switch (State) {

        // 通常状態
        case 0:

            // 画像を元の大きさで表示する
            W=H=1;

            // X座標の更新
            X+=VX;

            // 画面からはみ出したら、初期化して再び出現させる
            if (X<-1 || X>MAX_X) Init();
```

```
    // 穴に落ちたかどうかの判定処理
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==1 &&
            abs(X-mover->X)<max_dist &&
            abs(Y-mover->Y)<max_dist
        ) {
            Hole=(CManualHole*)mover;
            if (Hole->Size==1 && !Hole->Enemy) {

                // 敵が穴に落ちたら、
                // X座標を穴のX座標に合わせてから、
                // 穴落ち状態に移行する
                // また、1つの穴に1つの敵だけを落とすため、
                // 穴のEnemyメンバに、
                // 落ちた敵のポインタを記録しておく
                X=Hole->X;
                Y=Hole->Y;
                Hole->Enemy=this;
                State=1;
                break;
            }
        }
    }
    break;

// 穴落ち状態
case 1:

    // 穴のサイズに合わせて、敵の画像を縮小表示する
    W=H=Hole->Size;

    // 敵を回転させて、穴に落ちていく様子を表現する
    Angle+=fall_vangle;

    // 回転角度が一定値に達したら、
    // 敵が完全に穴に落ちたと判定する
    // 時間を設定して、待機状態に移行する
    if (Angle>=0.5f) {
        Time=fall_time;
        State=2;
    }
    break;

// 待機状態
case 2:

    // 穴のサイズに合わせて、敵の画像を縮小表示する
    W=H=Hole->Size;
```

## List 4-6

```
            // 残り時間を減らし、
            // 時間が0になったら穴抜け状態に移行する
            Time--;
            if (Time==0) State=3;
            break;

        // 穴抜け状態
        case 3:

            // 穴のサイズに合わせて、敵の画像を縮小表示する
            W=H=Hole->Size;

            // 敵を回転させて、穴から抜け出す様子を表現する
            Angle-=fall_vangle;

            // 回転角度が一定値に達したら、
            // 敵が完全に穴から出たと判定する
            // X座標を設定して、通常状態に移行する
            // 出た穴にまたすぐに落ちないために、
            // X座標は穴の真上から少しずらしている
            // また、穴のEnemyメンバにNULLを書き込んで、
            // 落ちた敵の記録を消去する
            if (Angle<=0) {
                Angle=0;
                X=Hole->X+max_dist*VX/abs(VX);
                Hole->Enemy=NULL;
                State=0;
            }
            break;
    }

    // 穴に埋まったかどうかの判定処理
    // 敵が穴落ち状態・待機状態・穴抜け状態のいずれかのときに、
    // 穴が完全に埋まったら、敵が穴に埋まったと判定する
    // このサンプルでは敵を初期化して再び出現させる
    if (State!=0 && Hole->Size==0) Init();

    return true;
}

// キャラクターの移動処理を行うMove関数
bool CManualHoleMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 穴との当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=1.0f;
```

```
// レバーの入力に応じて上下左右に移動する
// 穴を掘ったり埋めたりするときのために、
// キャラクターが移動した方向を保存しておく
// VXとVYはキャラクターの速度を表す変数
// DirXとDirYはキャラクターの移動方向を表す変数
VX=VY=0;
if (is->Left) {
    DirX=-1;
    DirY=0;
    VX=-speed;
} else
if (is->Right) {
    DirX=1;
    DirY=0;
    VX=speed;
} else
if (is->Up) {
    DirX=0;
    DirY=-1;
    VY=-speed;
} else
if (is->Down) {
    DirX=0;
    DirY=1;
    VY=speed;
}

// X座標を更新し、キャラクターが画面からはみ出さないように補正する
X+=VX;
if (X<0) X=0;
if (X>MAX_X-1) X=MAX_X-1;

// Y座標を更新し、キャラクターが画面からはみ出さないように補正する
Y+=VY;
if (Y<0) Y=0;
if (Y>MAX_Y-1) Y=MAX_Y-1;

// 穴との当たり判定処理
// 穴は通り抜けられないので、
// 穴に接触したら、X座標とY座標の更新をキャンセルする
for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
    CMover* mover=(CMover*)i.Next();
    if (
        mover->Type==1 &&
        abs(X-mover->X)<max_dist &&
        abs(Y-mover->Y)<max_dist
    ) {
        X-=VX;
        Y-=VY;
```

**List 4-6**

```
			break;
		}
	}

	// キャラクターが移動してきた方向を考慮して、
	// キャラクターの前方に穴があるかどうかを調べる
	// ここでは前方の座標（X+DirX, Y+DirY）が、
	// 穴から一定範囲内にあるかどうかを調べている
	CManualHole* hole=NULL;
	for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
		CMover* mover=(CMover*)i.Next();
		if (
			mover->Type==1 &&
			abs(X+DirX-mover->X)<max_dist &&
			abs(Y+DirY-mover->Y)<max_dist
		) {
			hole=(CManualHole*)mover;
			break;
		}
	}

	// ボタン0を押したときに穴を掘る処理
	if (!PrevButton && is->Button[0]) {

		// 前方に穴がある場合には、その穴を広げる
		if (hole) {
			hole->Dig();
		} else

		// 前方に穴がない場合には、新しい穴を掘る
		{
			new CManualHole(X+DirX, Y+DirY, 1);
		}
	}

	// ボタン1を押したときに穴を埋める処理
	if (!PrevButton && is->Button[1]) {

		// 前方に穴がある場合には、その穴を埋める
		if (hole) hole->Fill();
	}

	// ボタンを押した瞬間を判定するために、
	// 現在のボタンの状態を保存しておく
	PrevButton=is->Button[0]||is->Button[1];

	// X方向の速度に応じて、キャラクターを傾けて表示する
	Angle=VX/speed*0.1f;

	return true;
}
```

SAMPLE

「MANUAL HOLE」は手動穴のサンプルです。レバーでキャラクターを上下左右に移動することができます。ボタン0を押すと、キャラクターの進行方向に穴を掘ることができ、ボタン1を押すと穴を埋めます。敵が落ちた状態で穴を埋めると、敵を倒すことができます。

**MANUAL HOLE** → p. 396

## ロープを張る

ステージ内の好きな場所にロープを張るアクションです。張ったロープをつたって、高い場所に登ったり、谷間を越えたりすることができます。

Fig. 4-51のような場所にロープを張ることを考えましょう。キャラクターは右を向いているとします。

ボタンを押すと、キャラクターがロープを出します(Fig. 4-52)。ロープはキャラクターの斜め上方向に出ることにしましょう。ここではキャラクターが右を向いているので、右上にロープが出ます。出たロープは、壁や天井に当たるまで伸びます。壁や天井に当たるとロープの伸びは止まって、ロープが張られた状態になります。

キャラクターはロープをつたって移動することができます(Fig. 4-53)。上手にロープを張れば、ロープをつたって高い場所に登ることが可能です(Fig. 4-54)。ロープから飛び降りるには、レバーを下に入力します。

ロープを張るアクションを採用したゲームには、例えば「ロックンロープ」があります。このゲームでは、段差のあるステージにロープを張って、上の階に登っていきます。敵につかまらないようにロープをつたっていくのですが、自分が張ったロープを敵がつたってくることも

**Fig. 4-51** ロープを張る前の状態

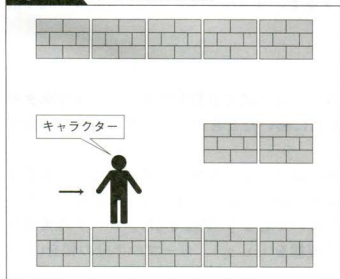



**Fig. 4-52** ロープを出す

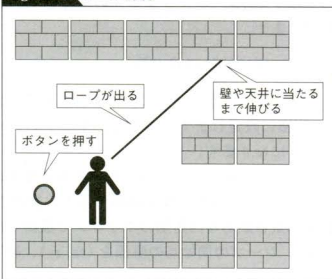



Fig. 4-53 ロープをつたって移動する

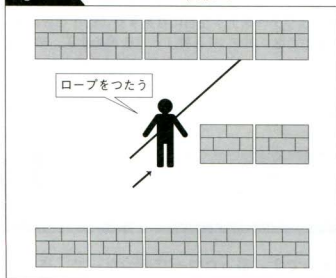
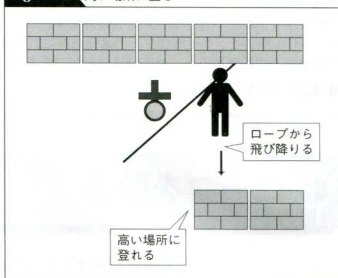



Fig. 4-54 高い場所に登る

あるので、慎重なプレイが要求されます。

「海腹川背」でもロープを張ることができます。このゲームでは、ロープが自在に伸縮するうえに、遠心力や張力が働くため、ロープを利用した非常に幅広いアクションが可能です。高い場所に登るだけではなく、振り子のようにぶら下がったり、張力を使ってジャンプしたりダッシュしたりと、実にさまざまな動きが楽しめます。

## アルゴリズム Algorithm

ロープを張るアクションを実現するときのポイントは、キャラクターとロープの当たり判定処理です（Fig. 4-55）。ロープの当たり判定は、ロープに沿った平行四辺形になっています。図に点線で示した範囲にキャラクターが接触したら、ロープにつかまったと判定します。

ロープにつかまっているときには、キャラクターのY座標を調整することによって、キャラクターをロープに沿って移動させます（Fig. 4-56）。ロープの開始点の座標を（rx, ry）、ロープの角度をAngle、キャラクターの座標を（X, Y）とすると、キャラクターのY座標は次の式で計算できます。

```
Y=ry+(X-rx)/tan(Angle)
```

キャラクターがロープにつかまっているときには、上記の式で計算した値を、キャラクターのY座標に設定します。すると、キャラクターが左右に動いたときに、ロープに沿って移動するようになります。

ロープを描画する方法はいろいろありますが、例えばポリゴンを使って描画することもできます。ロープを1本の線で描画すると、画面の解像度が高いと細くて見づらくなるので、ある程度の太さがある四角形のポリゴンで描画した方がよいでしょう（Fig. 4-57）。ロープの角度をAngle、ロープの開始点の座標を（rx, ry）、終了点の座標を（sx, sy）、ロープの半分の幅をtとすると、ポリゴンの四隅の座標は次のようになります。

```
(rx-sin(Angle)*t, ry-cos(Angle)*t)
(rx+sin(Angle)*t, ry+cos(Angle)*t)
(sx-sin(Angle)*t, sy-cos(Angle)*t)
(sx+sin(Angle)*t, sy+cos(Angle)*t)
```

**Fig. 4-55** ロープの当たり判定

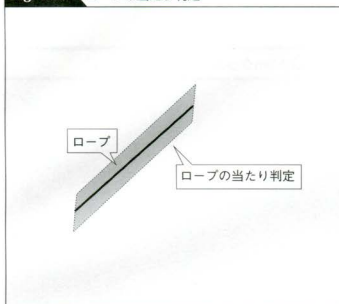
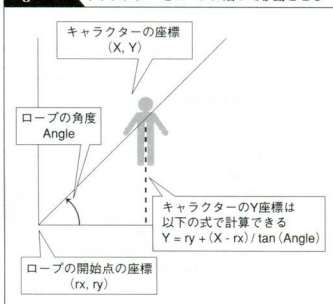
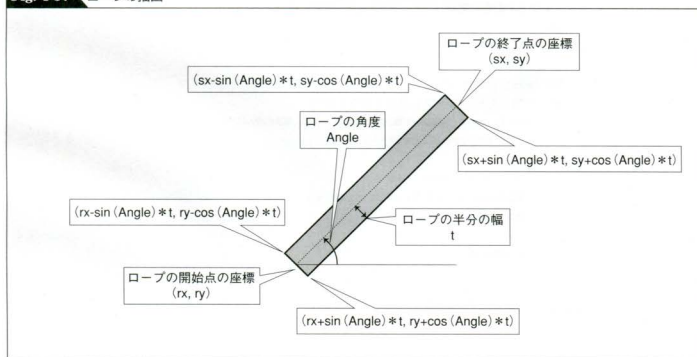



**Fig. 4-56** キャラクターをロープに沿って移動させる

**Fig. 4-57** ロープの描画

## プログラム Program

List 4-7はロープを張るアクションのプログラムです。このサンプルでは、画面上に出るロープは1本にしました。新しいロープを出したときには、画面上にあった既存のロープは消えます。また、ロープを出す角度は固定になっていますが、レバーやボタンの操作で角度を調整できるようにしても面白いでしょう。

**List 4-7** ロープを張る(CPlaceRopeクラス、CPlaceRopeManクラス)

```
// ロープの移動処理を行うMove関数
bool CPlaceRope::Move(const CInputState* is) {

    // 壁との当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=0.5f;

    // 状態に応じて分岐する
    // Stateはロープの状態を表す
    switch (State) {

        // ロープが伸びている状態
        case 0:

            // X座標とY座標の更新
            X+=VX;
            Y+=VY;

            // 壁に接触したかどうかの判定処理
            // 壁に接触したら、ロープを伸ばすのを止める
            for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
                CMover* mover=(CMover*)i.Next();
                if (
                    mover->Type==1 &&
                    abs(mover->X-X)<max_dist &&
                    abs(mover->Y-Y)<max_dist
                ) {
                    State=1;
                }
            }
            break;
    }

    return true;
}
```

```
// ロープの描画処理を行うDraw関数
void CPlaceRope::Draw() {

    // 頂点の計算に必要な値を用意する
    float
        w=Game->GetGraphics()->GetWidth()/MAX_X,
        h=Game->GetGraphics()->GetHeight()/MAX_Y,
        rad=Angle*D3DX_PI*2,
        c=cosf(rad)*0.05f,
        s=sinf(rad)*0.05f;

    // 各頂点のX座標の計算
    float x[]={
        (RootX-s+0.5f)*w, (X-s+0.5f)*w,
        (RootX+s+0.5f)*w, (X+s+0.5f)*w
    };

    // 各頂点のY座標の計算
    float y[]={
        (RootY-c+0.5f)*h, (Y-c+0.5f)*h,
        (RootY+c+0.5f)*h, (Y+c+0.5f)*h
    };

    // ポリゴンを描画する
    // Game->Texture[TEX_FILL]はベタ塗りのテクスチャ
    Game->Texture[TEX_FILL]->Draw(
        x[0], y[0], Color, 0, 0,
        x[1], y[1], Color, 0, 0,
        x[2], y[2], Color, 0, 0,
        x[3], y[3], Color, 0, 0
    );
}

// キャラクターの移動処理を行うMove関数
bool CPlaceRopeMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 落下時の最大スピード
    float jump_speed=-0.4f;

    // 落下中の加速度
    float jump_accel=0.02f;

    // ロープを打ち出す角度
    float rope_angle=0.15f;

    // ロープまたは床との当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
```

## List 4-7

```
float max_dist=1.0f;

// レバーの入力に応じて左右に移動する
// ロープを打ち出すときのために、
// キャラクターが移動した方向を保存しておく
// VXとVYはキャラクターの速度を表す変数
// DirXとDirYはキャラクターの移動方向を表す変数
VX=0;
if (is->Left) {
    DirX=-1;
    VX=-speed;
}
if (is->Right) {
    DirX=1;
    VX=speed;
}

// X座標の更新
X+=VX;

// 床との当たり判定処理
// 床に接触したら、
// 床にキャラクターがちょうど接するようにX座標を調整する
for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
    CMover* mover=(CMover*)i.Next();
    if (
        mover->Type==1 &&
        abs(mover->X-X)<max_dist &&
        abs(mover->Y-Y)<max_dist
    ) {
        X=mover->X+(X<mover->X?-max_dist:max_dist);
    }
}

// Y方向の速度を更新し、落下スピードが一定値を超えないように補正する
VY+=jump_accel;
if (VY>-jump_speed) VY=-jump_speed;

// Y座標の更新
Y+=VY;

// ロープにつかまって移動する処理
// レバーを下に入れていないときには、ロープにつかまる
// ロープにつかまっているときにレバーを下に入れると、
// ロープから離れて落下する
if (!is->Down) {
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (mover->Type==2) {
            CPlaceRope* rope=(CPlaceRope*)mover;
```

```
                // キャラクターのX座標に応じて、
                // ロープにつかまったときのY座標を計算する
                float rope_y=
                    rope->RootY+
                    (rope->RootX-X)/tanf(rope->Angle*D3DX_PI*2);

                // ロープに接触しているかどうかを判定する
                // 接触しているときには、キャラクターのY座標を調整して、
                // ロープに沿って移動させる
                if (
                    (rope->RootX<=X && X<=rope->X ||
                    rope->X<=X && X<=rope->RootX) &&
                    abs(rope_y-Y)<max_dist
                ) {
                    Y=rope_y;
                    break;
                }
            }
        }
    }

    // ボタンを押したときにロープを打ち出す
    if (!PrevButton && is->Button[0]) {

        // 画面内にある既存のロープを消去する
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (mover->Type==2) i.Remove();
        }

        // 新しいロープを生成する
        new CPlaceRope(X, Y, DirX*rope_angle);
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // 床との当たり判定処理
    // 床に接触したら、
    // 床にキャラクターがちょうど接するようにY座標を調整する
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==1 &&
            abs(mover->X-X)<max_dist &&
            abs(mover->Y-Y)<max_dist
        ) {
            Y=mover->Y+(Y<mover->Y?-max_dist:max_dist);
```

**List 4-7**

```
        }
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「SETTING ROPE」はロープを張るアクションのサンプルです。レバーでキャラクターを左右に動かし、ボタンで進行方向斜め上に向かってループを発射します。ロープは床か天井にぶつかるまで伸び続けます。張られたロープをつたって移動することができます。

**SETTING ROPE** → p. 396

# 足場を作る

キャラクターが自分で足場を作るアクションです。作った足場に乗って、高い場所に移動できます。また、足場に乗った状態でさらに別の足場を作り、またその足場に乗って…という手順を繰り返すことによって、ずっと高いところまで登ることができます。

ボタンを押すと、キャラクターの前方に足場を作ることができます(Fig. 4-58)。ここでは虹の形をした足場を作ることにしました。

足場に向かって歩くと、足場に乗ることができます(Fig. 4-59)。足場は虹の形をしているので、キャラクターは虹のアーチに沿って滑らかに登っていきます。

足場に乗った状態でボタンを押すと、さらに別の足場を作ることができます(Fig. 4-60)。こ

**Fig. 4-58** 足場を作る

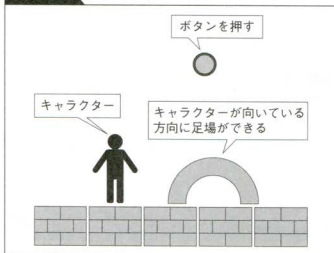
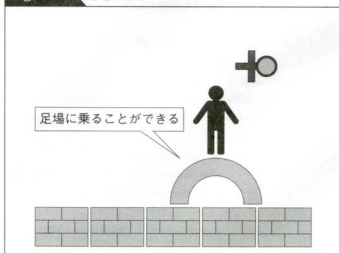



**Fig. 4-59** 足場に乗る

の手順を繰り返すと、足場をつなげて高いところに登ったり、広い谷間を越えたりすることが可能です（Fig. 4-61）。

一定時間が経過すると、足場は消えてしまいます（Fig. 4-62）。足下の足場がなくなると、キャラクターは落下します。

足場を作るアクションを採用したゲームには、「レインボーアイランド」などがあります。このゲームでは、ボタン操作で虹を作ることができます。虹を足場として使って、ステージを登っていくことがゲームの目的です。また、虹が接触したアイテムは主人公が取得したことになります。さらに、虹の上にジャンプして乗ると虹を落とすことができます。落下中の虹を使って敵を攻撃したり、アイテムを取ったりと、足場として以外にもいろいろな使い方ができることが、このゲームの虹の特徴です。

Fig. 4-60 別の足場を作る

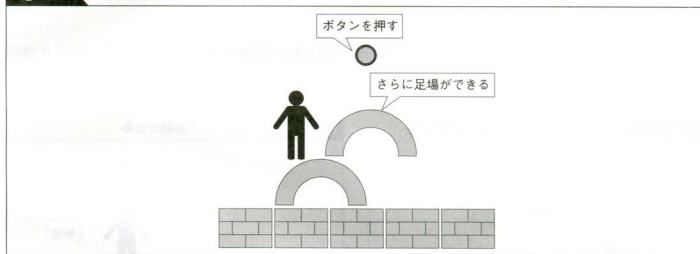



Fig. 4-61 足場をつなげて移動する

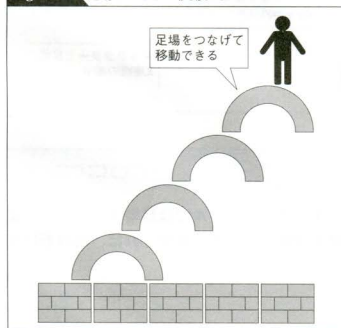



Fig. 4-62 足場が消える

## アルゴリズム　Algorithm

足場を作るアクションを実現するには、まず足場を作る処理が必要です。足場はキャラクターの前方に作ります。キャラクターの前方がどの方向なのかを決めるためには、キャラクターが進んできた向きを保存しておく必要があります。

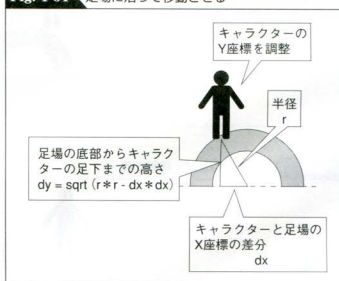
次に、キャラクターと足場の当たり判定処理が必要です(Fig. 4-63)。足場が虹のようなアーチ形の場合には、アーチに沿った当たり判定を用意して、キャラクターが当たり判定の範囲に入ったかどうかを調べます。

キャラクターが足場に接触していたら、キャラクターのY座標を調整することによって、足場に沿って移動させます(Fig. 4-64)。足場の半径をr、キャラクターと足場のX座標の差分をdxとすると、足場の底部からキャラクターの足下までの高さdyは、

```
dy=sqrt(r*r-dx*dx)
```

のように計算できます。当たり判定処理を行うときにも、同様の計算でアーチ形の当たり判定を作るとよいでしょう。

**Fig. 4-63** キャラクターと足場の当たり判定処理

**Fig. 4-64** 足場に沿って移動させる

## プログラム　Program

List 4-8は足場を作るアクションのプログラムです。このサンプルでは足場を虹の形にしましたが、当たり判定処理と、キャラクターのY座標を調整する処理を変更すれば、別の形にすることもできます。

**List 4-8** 足場を作る(CFootholdクラス、CMakeFootholdManクラス)

```
// 足場の移動処理を行うMove関数
// 残り時間を減少させ、時間が0になったら足場を消去する
// 本書のサンプルでは、Move関数でfalseを返すと、
// 呼び出し元の関数がオブジェクトを消去してくれる
bool CFoothold::Move(const CInputState* is) {
    Time--;
    return Time>0;
}

// キャラクターの移動処理を行うMove関数
bool CMakeFootholdMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // ジャンプの初速度
    float jump_speed=-0.4f;

    // ジャンプ中の加速度
    float jump_accel=0.02f;

    // 床との当たり判定処理を行うための定数
    // X座標の差分の最大値、Y座標の差分の最小値と最大値
    float floor_max_x=0.6f;
    float floor_min_y=0.5f;
    float floor_max_y=1.0f;

    // 足場との当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float foothold_max_x=2.0f;
    float foothold_min_y=0.0f;

    // 足場の半径
    float foothold_r=2.0f;

    // レバーの入力に応じて左右に移動する
    // 足場を作るときのために、
    // キャラクターが移動した方向を保存しておく
    // VXとVYはキャラクターの速度を表す変数
    // DirXとDirYはキャラクターの移動方向を表す変数
    VX=0;
    if (is->Left) {
        DirX=-1;
        VX=-speed;
    }
    if (is->Right) {
        DirX=1;
        VX=speed;
```

List 4-8

```
}

// ボタンを押したら足場を生成する
if (!PrevButton && is->Button[0]) {
    new CFoothold(X+DirX*2, Y+0.5f, 2);
}

// X座標を更新し、キャラクターが画面からはみ出さないように補正する
X+=VX;
if (X<0) X=0;
if (X>MAX_X-1) X=MAX_X-1;

// Y方向の速度を更新し、落下スピードが一定値を超えないように補正する
VY+=jump_accel;
if (VY>-jump_speed) VY=-jump_speed;

// Y座標を更新し、キャラクターが画面からはみ出さないように補正する
Y+=VY;
if (Y<0) Y=0;
if (Y>MAX_Y-1) Y=MAX_Y-1;

// 床または足場との当たり判定処理
for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
    CMover* mover=(CMover*)i.Next();

    // 床との当たり判定処理
    // 床に接触した場合には、
    // キャラクターが床に乗るようにY座標を調整する
    if (mover->Type==1) {
        if (
            abs(mover->X-X)<floor_max_x &&
            mover->Y-Y>floor_min_y &&
            mover->Y-Y<floor_max_y
        ) {
            Y=mover->Y-floor_max_y;
        }
    }

    // 足場との当たり判定処理
    // 足場に接触した場合には、
    // キャラクターが足場に乗るようにY座標を調整する
    // 足場はアーチ形なので、
    // アーチに沿った当たり判定を用意し、
    // キャラクターもアーチに乗るようにY座標を設定する
    if (mover->Type==2) {
        float dx=abs(mover->X-X);
        if (dx<foothold_r) {
            float foothold_max_y=
                foothold_min_y+sqrt(foothold_r*foothold_r-dx*dx);
            if (
```

```
                        mover->Y-Y>foothold_min_y &&
                        mover->Y-Y<foothold_max_y
                    ) {
                        Y=mover->Y-foothold_max_y;
                    }
                }
            }
        }

        // ボタンを押した瞬間を判定するために、
        // 現在のボタンの状態を保存しておく
        PrevButton=is->Button[0];

        // X方向の速度に応じて、キャラクターを傾けて表示する
        Angle=VX/speed*0.1f;

        return true;
    }
```

SAMPLE

「FOOTHOLD」は足場を作るアクションのサンプルです。レバーでキャラクターを左右に移動することができます。ボタンを押すと、進行方向に虹の形の足場を作ることができます。キャラクターは足場の上に乗ることができ、足場の上からさらに足場を作ることもできます。足場は一定時間が経過すると消えます。

**FOOTHOLD** → p. 396

## ⊕壁を倒す

垂直に立っている壁を押すことで倒すアクションです。敵が壁の向こうにいるときにタイミングよく壁を倒すと、壁で敵をつぶすこともできます。

ステージには垂直に立った壁が配置されています(Fig. 4-65)。壁を押すようにキャラクターを移動させると、キャラクターが移動した方向に壁が倒れます(Fig. 4-66)。右に押せば壁は右に倒れ、左に押せば左に倒れます。

壁の向こうに敵がいるときには、タイミングよく敵に向かって壁を倒します(Fig. 4-67)。うまく倒せば、敵を壁の下敷きにすることができます(Fig. 4-68)。

ゲームによっては、壁にもう1つの使い方があります。壁を倒したあとにボタンを押します(Fig. 4-69)。すると、すべての壁がいっせいに起き上がります(Fig. 4-70)。

この動きを利用して敵を攻撃するには、倒れた壁に敵が乗っているときにボタンを押します(Fig. 4-71)。

Fig. 4-65 垂直に立った壁

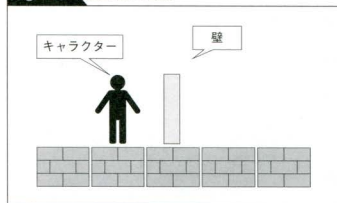



Fig. 4-66 壁を倒す

Fig. 4-67 敵に向かって壁を倒す

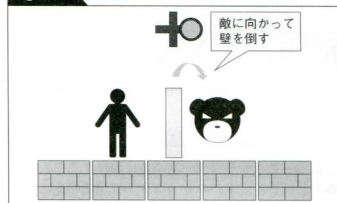



Fig. 4-68 敵を壁の下敷きにする

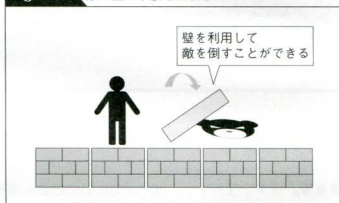



Fig. 4-69 壁を倒したあとにボタンを押す

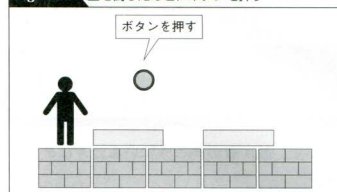



Fig. 4-70 壁が起き上がる

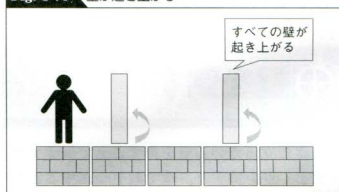



Fig. 4-71 敵が壁に乗っているときにボタンを押す

Fig. 4-72 敵を跳ね飛ばす

タイミングがよければ、壁が上がる勢いで敵を跳ね飛ばすことができます(Fig. 4-72)。複数の壁にそれぞれ敵が乗っているときには、複数の敵を同時に跳ね飛ばすことも可能です。

壁を倒すアクションを採用したゲームには、例えば「ぺったんピュー」があります。このゲームでは、ステージに立っている壁を押して倒し、敵を壁の下敷きにすることができます。また、倒した壁をボタンで起こし、壁の上にいる敵を跳ね飛ばすことも可能です。

「ロンパーズ」でも壁を倒すことができます。このゲームには壁を起こすアクションはありませんが、短い壁や長い壁など、いろいろなサイズの壁があることが特徴です。

## アルゴリズム Algorithm

壁を倒すアクションを実現するには、まずキャラクターと壁の当たり判定処理が必要です(Fig. 4-73)。壁の周囲の一定範囲内にキャラクターがいたら、壁を押していると判定します。

キャラクターが壁を押したら、押された向きに壁を倒します。壁が押された向きは、キャラクターの移動方向と同じです。

壁には次のような4つの状態があります(Fig. 4-74)。壁の状態に応じて、異なる処理を行います。

Fig. 4-73 キャラクターと壁の当たり判定処理

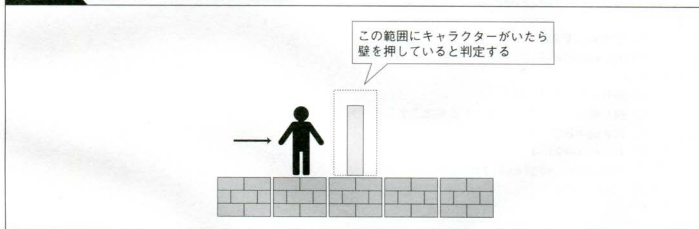
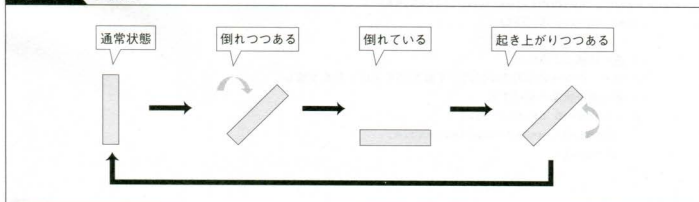



Fig. 4-74 壁の状態

・通常状態　　：壁が垂直に立っている状態。キャラクターが壁を押すと、倒れ途中状態へ移行する
・倒れ途中状態：壁が倒れていく途中の状態。完全に倒れると、倒れ状態へ移行する
・倒れ状態　　：壁が完全に倒れた状態。ボタンを押すと、起き途中状態へ移行する
・起き途中状態：壁が起きていく途中の状態。完全に起きると、通常状態へ移行する

敵がつぶされるか跳ね飛ばされるかは、壁の状態によって決まります。倒れ途中状態の壁に接触すると、敵はつぶされます。起き途中状態の壁に接触すると、敵は跳ね飛ばされます。

## プログラム Program

List 4-9は壁を倒すアクションのプログラムです。壁の状態と敵の状態をそれぞれ管理して、状態に応じた処理を行うことがポイントです。壁については、倒れるときには比較的ゆっくりと、起き上がるときには素早く動くようにしています。

**List 4-9** 壁を倒す（CFlappingPanelクラス、CFlappingPanelEnemyクラス、CFlappingPanelManクラス）

```
// 壁を起こすGetUp関数
void CFlappingPanel::GetUp() {

    // 壁を起こすスピード
    float vangle=0.125f;

    // 倒れ状態のときには、
    // 壁が倒れている方向に応じて壁を起こすスピードを設定し、
    // 起き途中状態へ移行する
    if (State==2) {
        VAngle=(Angle<0)?vangle:-vangle;
        State=3;
    }
}

// 壁を倒すGetDown関数
void CFlappingPanel::GetDown(float vx) {
    float vangle=0.05f;

    // 通常状態のときには、
    // キャラクターの移動方向に応じて壁を倒すスピードを設定し、
    // 倒れ途中状態へ移行する
    if (State==0 && vx!=0) {
        VAngle=(vx<0)?-vangle:vangle;
        State=1;
    }
}
```

```
// 壁の移動処理を行うMove関数
bool CFlappingPanel::Move(const CInputState* is) {

    // 状態に応じて分岐する
    // Stateは壁の状態を表す
    switch (State) {

        // 倒れ途中状態
        case 1:

            // 角度の更新
            Angle+=VAngle;

            // 角度が90度に近づいたら、倒れ状態へ移行する
            // このサンプルでは角度を0（0度）から1（360度）で表す
            // 0.25は90度に相当する
            if (abs(Angle)>=0.25f) State=2;
            break;

        // 起き途中状態
        case 3:

            // 角度の更新
            Angle+=VAngle;

            // 角度が0度に近づいたら、
            // 角度を0度にして、通常状態へ移行する
            if (abs(Angle)<0.1f) {
                Angle=0;
                State=0;
            }
            break;
    }

    return true;
}

// 敵の移動処理を行うMove関数
bool CFlappingPanelEnemy::Move(const CInputState* is) {

    // パネルとの当たり判定処理を行うための定数
    // X座標の差分の最大値
    float max_dist=1.6f;

    // 壁につぶされたときに、敵がつぶれる速さ
    float shrink=0.2f;

    // 壁に跳ね飛ばされたときに、
    // 敵に設定するX方向とY方向の初速度
    float jump_vx=-0.3f;
```

**List 4-9**

```
float jump_vy=-0.6f;

// 空中の敵にかかる加速度
float jump_accel=0.02f;

// 状態に応じて分岐する
// Stateは敵の状態を表す
switch (State) {

    // 通常状態
    case 0:

        // X座標の更新
        X+=VX;

        // 敵が画面からはみ出したら、初期化して再び出現させる
        if (X<-1) X=MAX_X;

        // 壁との当たり判定処理
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (
                mover->Type==1 &&
                abs(mover->X-X)<max_dist
            ) {
                CFlappingPanel* panel=(CFlappingPanel*)mover;

                // 壁が左右どちらに倒れているかによって、
                // 当たり判定の位置を変える
                if (
                    panel->Angle>0 && panel->X<X ||
                    panel->Angle<0 && X<panel->X
                ) {

                    // 倒れ状態の壁に接触したら、敵をつぶす
                    // つぶれ状態へ移行する
                    if (panel->State==1) {
                        State=1;
                        break;
                    }

                    // 起き状態の壁に接触したら、敵を跳ね飛ばす
                    // 初速度を設定して、跳ね飛び状態へ移行する
                    if (panel->State==3) {
                        VX=jump_vx*panel->Angle/abs(panel->Angle);
                        VY=jump_vy;
                        Angle=0.5f;
                        State=2;
                        break;
                    }
```

```
                    break;
                }
            }
        }
        break;

    // つぶれ状態
    case 1:

        // 敵をだんだんつぶしていく
        H-=shrink;
        Y+=shrink*0.5f;

        // 完全につぶれたら、初期化して再び出現させる
        if (H<=0) Init();
        break;

    // 跳ね飛び状態
    case 2:

        // X座標・速度・Y座標の更新
        X+=VX;
        VY+=jump_accel;
        Y+=VY;

        // 画面の左右端からはみ出したら、初期化して再び出現させる
        if (X<-1 || X>MAX_X) Init();
        break;
    }

    return true;
}

// キャラクターの移動処理を行うMove関数
bool CFlappingPanelMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 壁との当たり判定処理を行うための定数
    // X座標の差分の最大値
    float max_dist=0.7f;

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
```

**List 4-9**

```
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // パネルとの当たり判定処理
    // 通常状態のパネルに接触したら、パネルを押して倒す
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==1 &&
            abs(X-mover->X)<max_dist
        ) {
            CFlappingPanel* panel=(CFlappingPanel*)mover;
            panel->GetDown(VX);
        }
    }

    // ボタンを押したときにパネルを起こす処理
    // 倒れ状態のすべてのパネルを起こす
    if (!PrevButton && is->Button[0]) {
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (mover->Type==1) {
                CFlappingPanel* panel=(CFlappingPanel*)mover;
                panel->GetUp();
            }
        }
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「FLAPPING PANEL」は壁を倒すアクションのサンプルです。レバーでキャラクターを左右に移動することができます。壁の上をキャラクターで通過すると、壁を倒すことができます。壁は通過した向きに合わせて倒れます。タイミングよく壁を倒すことで、敵を下敷きにすることができます。また、ボタンを押すと、倒れた壁がいっせいに起き上がります。壁で敵を跳ね飛ばすこともできます。

**FLAPPING PANEL** → p. 396

# 地面を落とす

地面にひびを入れることによって分断し、地面の一部を落として消滅させるアクションです。雰囲気としては、島に亀裂を入れて海中に沈下させるような様子になります。ひびを入れることによって、地面をいろいろな形に切り取って落とすことができる点が、このアクションの面白いところです。

ステージの地面には杭が並んでいます(Fig. 4-75)。地面にひびを入れるには、この杭に接触してボタンを押します。すると、キャラクターが向いている方向にひびが入ります(Fig. 4-76)。

地面にひびを入れていくと、地面全体が2つの領域に分かれます(Fig. 4-77)。すると、分かれた領域のうち、面積の小さい方が落ちてなくなります(Fig. 4-78)。このアクションが面白いのは、多くのひびを入れて複雑な形の領域を作ったときにも、面積の小さい方の領域が正確に

**Fig. 4-75** 杭が並んだ地面

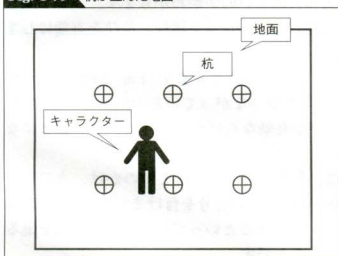



**Fig. 4-76** 杭に接触してボタンを押す

**Fig. 4-77** 地面が2つの領域に分かれる

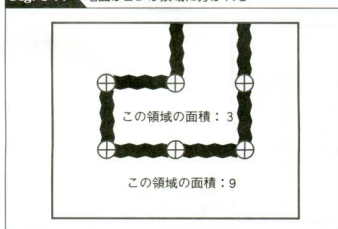
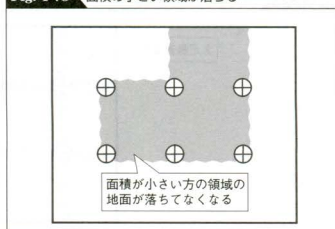



**Fig. 4-78** 面積の小さい領域が落ちる

落ちることです。

地面を落とすアクションを採用したゲームには、「ディグダグII」があります。このゲームの舞台は島です。島にある杭の上でボタンを押すと、島にひびを入れることができます。ひびで島が2つの領域に分断されると、狭い方の領域が海中に沈んでなくなります。うっかり自分のキャラクターがいる領域を落とすとミスになるので、慎重にひびを入れる必要があります。

このゲームでは、島が落ちるときに敵をいっしょに落とすと、敵を倒すことができます。敵をポンプでふくらませて動けなくさせておき、その敵がいる領域を落として敵を倒す、といった戦術も使えます。

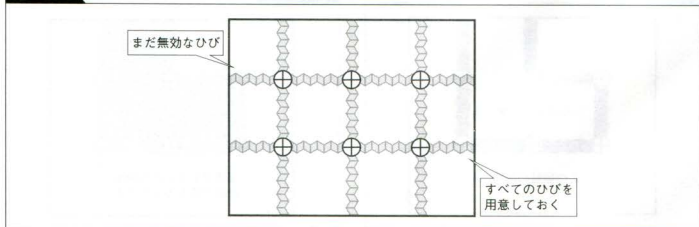
## アルゴリズム Algorithm

地面を落とすアクションを実現するときのポイントは、ひびや杭のデータ構造です。まず、ステージにはあらかじめすべてのひびを配置しておきます (Fig. 4-79)。最初はひびを無効にしておき、見えない状態にします。地面はひびによって格子状の区画に分割されています。

杭には、四方に接する4本のひびへのポインタを保持させます (Fig. 4-80)。こうすれば、キャラクターが杭に接触したときに、どこにひびを入れればよいのかがわかります。ひびを入れるときには、キャラクターの方向に応じて4本のひびのなかから1つを選び、ひびを有効にします。

ひびには、ひびの両側に接する地面2区画へのポインタを保持させます (Fig. 4-81)。こうすれば、ひびを調べることによって、地面のどの区画同士がつながっているのかがわかります。ひびが無効ならば2つの区画はつながっており、ひびが有効ならばつながっていないことになります。

地面にひびを入れて2つの領域に分割したときに、どちらの領域を落とすのかは、次のような手順で決めます。説明のため、ここでは区画に0から11までの番号を付けました。まずは0番地に着目して、この区画に色を塗ります (Fig. 4-82)。色を塗るといっても、画面上で色を塗るのではなくて、区画のデータにマークを付けるということです。

Fig. 4-79 ひびの配置

次に、ひびを調べて、0番地につながっている区画を探します。例えば1番地は、ひびをはさんで0番地と隣り合っています。このひびは無効なので、0番地と1番地はつながっています。そこで、1番地に0番地と同じ色を塗ります(Fig. 4-83)。

同様に、今度は1番地につながっている区画を探します。ひびをはさんで1番地と隣り合っている区画は、0番地・2番地・5番地です。このうち0番地にはすでに色が塗られているので、も

**Fig. 4-80** 杭はひびのポインタを保持する

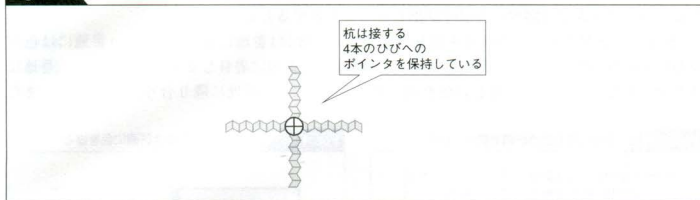



**Fig. 4-81** ひびは地面のポインタを保持する

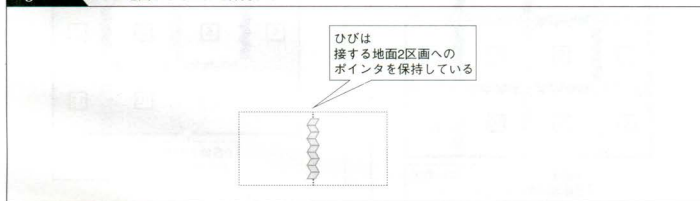



**Fig. 4-82** 0番地に色を塗る

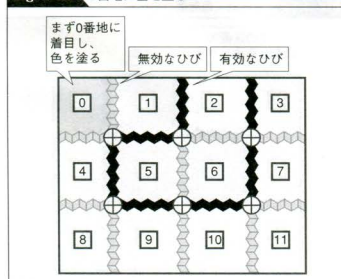
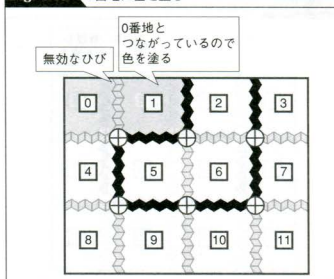



**Fig. 4-83** 1番地に色を塗る

う色は塗りません。また、2番地側および5番地側にあるひびは有効なので、1番地と2番地および5番地はつながっていません。この場合は、2番地と5番地の隣はもう調べません(Fig. 4-84)。

再び0番地に戻って、隣り合っている区画を探します。1番地以外に、0番地は4番地とも隣り合っています。0番地と4番地の間にあるひびは無効なので、これらの区画はつながっています。そこで、4番地に0番地と同じ色を塗ります。同様に、4番地に隣り合う区画を調べ、また区画に隣り合う区画を調べ…という過程を繰り返します。すると、スタート地点の0番地につながっている区画を、すべて同じ色で塗ることができます(Fig. 4-85)。区画を調べる過程の繰り返しは、プログラムでは関数の再帰呼び出しとして実装するとよいでしょう。

0番地につながるすべての区画を塗り終えたら、今度は1番地に着目します。1番地には色が塗られているので、ここでは何もしません。同様に、2番地に着目します(Fig. 4-86)。2番地はまだ塗られていないので、新しい色を塗ります。そして、2番地に隣り合う区画を調べ、また

**Fig. 4-84** 1番地に隣り合う区画を調べた結果

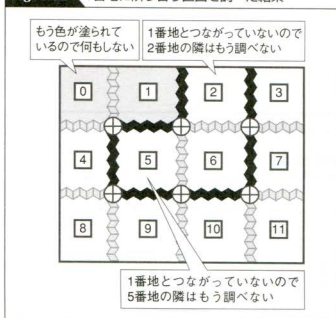



**Fig. 4-85** 0番地につながる全区画に色を塗る

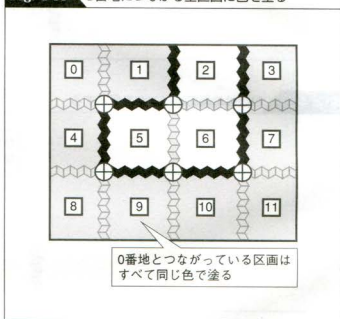



**Fig. 4-86** 2番地に色を塗る

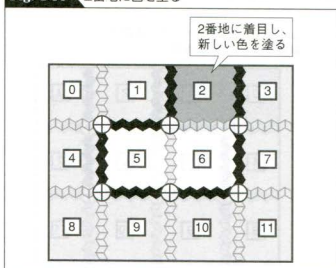
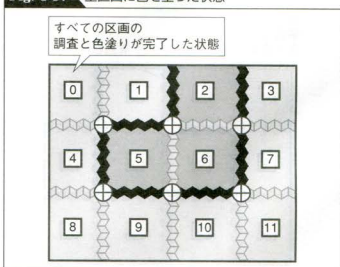



**Fig. 4-87** 全区画に色を塗った状態

その区画に隣り合う区画を調べ…という過程を繰り返して、2番地につながるすべての区画を同じ色で塗ります。

同じ要領で最後の11番地まで調べて、つながっている区画をそれぞれ同じ色で塗ります。すると、地面全体を2つの領域に塗り分けることができます(Fig. 4-87)。

地面全体に色を塗ったら、各々の色に塗られた区画を数えることによって、各領域の面積を調べます。最後に、面積が小さい方の領域を落とせば、地面を落とすアクションの完成です。

## プログラム Program

List 4-10は地面を落とすアクションのプログラムです。このサンプルでは地面全体が長方形ですが、ステージを生成するInit関数を変更すれば、さまざまな形のステージを作ることができます。

隣り合う区画を塗る処理は、MarkLand関数とDropLand関数で行います。MarkLand関数は自分自身を再帰的に呼び出して、ある区画につながっているすべての区画を同じ色で塗ります。

**List 4-10** 地面を落とす(CDropLandクラス、CDropLandCrackクラス、CDropLandStageクラス、CDropLandManクラス)

```
// 地面を落とすDrop関数
// 地面がまだ落ちていなければ落とす
// Stateは地面の状態を表す
void CDropLand::Drop() {
    if (State==0) State=1;
}

// 地面の移動処理を行うMove関数
bool CDropLand::Move(const CInputState* is) {

    // 地面が落ちるスピード
    float drop_speed=0.05f;

    // 地面を落とす処理
    if (State==1) {

        // 時間とともに穴の画像を大きくすることによって、
        // 地面が消える様子を表現する
        // 最初は穴を表示しないでおき、
        // 地面が落ちたときに暗い色の画像を表示することで、
        // 地面が落ちた後の様子を表現している
        W+=drop_speed;
        H+=drop_speed;

        // 画像の大きさが一定値に達したら、
        // 地面が完全に落ちたとして、
        // 地面を落とす処理を終える
```

List 4-10

```
        if (W>=1) {
            W=H=1;
            State=2;
        }
    }

    return true;
}

// ひびを有効にするCrack関数
// ひびが無効ならば有効にする
// Stateはひびの状態を表す
void CDropLandCrack::Crack() {
    if (State==0) State=1;
}

// ひびの移動処理を行うMove関数
bool CDropLandCrack::Move(const CInputState* is) {

    // ひびが出現するスピード
    float crack_speed=0.05f;

    // ひびを出現させる処理
    if (State==1) {

        // 時間とともにひびの画像を大きくすることによって、
        // ひびが入る様子を表現する
        W+=crack_speed;
        H+=crack_speed;

        // 画像の大きさが一定値に達したら、
        // ひびが完全に入ったとして、
        // ひびを入れる処理を終える
        if (W>=1) {
            W=H=1;
            State=2;
        }
    }

    return true;
}

// 地面の区画を塗るMarkLand関数
// 引数のlandは着目している区画を、markは色を表す
void CDropLandStage::MarkLand(CDropLand* land, int mark) {

    // ステージにあるひびの総数
    int crack_count=17;

    // 区画がまだ落ちていなくて、
```

```
	// かつ塗られていないかどうかを調べる
	if (land->State==0 && land->Mark<0) {

		// 区画を塗る
		// Markメンバは区画の色を表す
		land->Mark=mark;

		// すべてのひびを順番に処理して、
		// ひびをはさんで隣り合う区画を調べる
		for (int i=0; i<crack_count; i++) {

			// ひびがまだ無効の場合の処理
			// ひびの片側が着目している区画ならば、
			// ひびの反対側にある区画を処理する
			// MarkLand関数を再帰的に呼び出すことによって、
			// つながっているすべての区画について処理を行う
			if (Crack[i]->State==0) {
				if (Crack[i]->Land[0]==land) {
					MarkLand(Crack[i]->Land[1], mark);
				}
				if (Crack[i]->Land[1]==land) {
					MarkLand(Crack[i]->Land[0], mark);
				}
			}
		}
	}
}

// 地面を落とすDropLand関数
// ひびを1つ有効にするたびに、
// この関数を呼び出して、落ちる領域がないかどうかを調べる
void CDropLandStage::DropLand() {

	// ステージにある区画の総数
	int land_count=12;

	// すべての区画を塗っていない状態にする
	// Markメンバが-1の区画は塗られていないとして扱う
	for (int i=0; i<land_count; i++) Land[i]->Mark=-1;

	// 0番地から最後の番地まで、
	// 区画を順番に塗っていく
	// 各々の区画についてMarkLand関数を呼び出し、
	// つながっている区画を同じ色で塗る
	int mark=0;
	for (int i=0; i<land_count; i++) {
		if (Land[i]->State==0 && Land[i]->Mark<0) {
			MarkLand(Land[i], mark);
			mark++;
		}
```

**List 4-10**

```
    }

    // 地面が2つ以上の領域に塗り分けられたら、
    // 面積が小さな方の領域を落とす
    if (mark>=2) {

        // 最小の領域の色
        int min_mark=0;

        // 最小の領域の面積(区画の数)
        int min_count=land_count;

        // 各色について、その色に塗られた区画を数えて、
        // 最も区画の数が少ない色を調べる
        for (int j=0; j<mark; j++) {

            // 区画を数える
            int count=0;
            for (int i=0; i<land_count; i++) {
                if (Land[i]->Mark==j) count++;
            }

            // 区画の数が今までで一番少なければ、
            // 区画の数と色を記録する
            if (count<min_count) {
                min_count=count;
                min_mark=j;
            }
        }

        // 面積が小さな領域を落とす
        // その領域の色に塗られたすべての区画を落とす
        for (int i=0; i<land_count; i++) {
            if (Land[i]->Mark==min_mark) Land[i]->Drop();
        }
    }
}

// キャラクターの移動処理を行うMove関数
bool CDropLandMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 杭との当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=0.9f;

    // レバーの入力に応じて上下左右に移動する
    // ひびを入れるときのために、
```

```
// キャラクターが移動した方向を保存しておく
// VXとVYはキャラクターの速度を表す変数
// DirXとDirYはキャラクターの移動方向を表す変数
VX=VY=0;
if (is->Left) {
    DirX=-1;
    DirY=0;
    VX=-speed;
} else
if (is->Right) {
    DirX=1;
    DirY=0;
    VX=speed;
} else
if (is->Up) {
    DirX=0;
    DirY=-1;
    VY=-speed;
} else
if (is->Down) {
    DirX=0;
    DirY=1;
    VY=speed;
}

// X座標を更新し、キャラクターが画面からはみ出さないように補正する
X+=VX;
if (X<0) X=0;
if (X>MAX_X-1) X=MAX_X-1;

// Y座標を更新し、キャラクターが画面からはみ出さないように補正する
Y+=VY;
if (Y<0) Y=0;
if (Y>MAX_Y-1) Y=MAX_Y-1;

// 杭との当たり判定処理
// 杭は通り抜けられないので、杭に接触したら、
// X座標とY座標の更新をキャンセルする
for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
    CMover* mover=(CMover*)i.Next();
    if (
        mover->Type==1 &&
        abs(X-mover->X)<max_dist &&
        abs(Y-mover->Y)<max_dist
    ) {
        X-=VX;
        Y-=VY;
        break;
    }
}
```

**List 4-10**

```
    // ボタンを押したときにひびを入れる処理
    // 杭に接触してボタンを押したら、
    // キャラクターの向きに応じた方向にひびを入れる
    if (!PrevButton && is->Button[0]) {
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (
                mover->Type==1 &&
                abs(X+DirX-mover->X)<max_dist &&
                abs(Y+DirY-mover->Y)<max_dist
            ) {
                // キャラクターの向きに応じた方向にひびを入れる
                // 杭は四方にあるひびへのポインタを保持しているので、
                // キャラクターの向きに対応したポインタを使って、
                // ひびを有効にする
                CDropLandPile* pile=(CDropLandPile*)mover;
                if (DirX<0) pile->Crack[0]->Crack(); else
                if (DirX>0) pile->Crack[1]->Crack(); else
                if (DirY<0) pile->Crack[2]->Crack(); else
                if (DirY>0) pile->Crack[3]->Crack();

                // 地面を落とす処理を呼び出す
                Stage->DropLand();
                break;
            }
        }
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}

// ステージの初期化を行うInit関数
void CDropLandStage::Init() {

    // キャラクターの生成
    new CDropLandMan(this);

    // 地面の区画の生成
    // 12個の区画を生成する。
    for (int n=0, j=0; j<3; j++) {
        for (int i=0; i<4; i++, n++) {
            Land[n]=new CDropLand(i*4, j*4, 4, 4);
        }
```

```
    }

    // 縦方向のひびの生成
    // 8本のひびを生成する
    for (int n=0, j=0; j<2; j++) {
        for (int i=0; i<4; i++, n++) {
            Crack[n]=new CDropLandCrack(i*4+0.5f, j*4+3.5f, 3, 1, 0.25f);

            // ひびをはさんで隣り合う2つの区画へのポインタを登録する
            Crack[n]->Land[0]=Land[i+j*4];
            Crack[n]->Land[1]=Land[i+j*4+4];
        }
    }

    // 横方向のひびの生成
    // 9本のひびを生成する
    for (int n=0, j=0; j<3; j++) {
        for (int i=0; i<3; i++, n++) {
            Crack[n+8]=new CDropLandCrack(i*4+3.5f, j*4+0.5f, 1, 3, 0);

            // ひびをはさんで隣り合う2つの区画へのポインタを登録する
            Crack[n+8]->Land[0]=Land[i+j*4];
            Crack[n+8]->Land[1]=Land[i+j*4+1];
        }
    }

    // 杭の生成
    // 6本の杭を生成する
    for (int n=0, j=0; j<2; j++) {
        for (int i=0; i<3; i++, n++) {
            CDropLandPile* pile=new CDropLandPile(i*4+3.5f, j*4+3.5f, 1);

            // 杭の四方にあるひびへのポインタを登録する
            pile->Crack[0]=Crack[i+j*4];
            pile->Crack[1]=Crack[i+j*4+1];
            pile->Crack[2]=Crack[8+i+j*3];
            pile->Crack[3]=Crack[8+i+j*3+3];
        }
    }
}
```

SAMPLE

「DROPPING LAND」は地面を落とすアクションのサンプルです。レバーでキャラクターを上下左右に移動することができます。杭に接しているときにボタンを押すと、キャラクターの向きに合わせて地面にひびが入ります。ひびで地面を2つに区切ると、面積の小さい方が塗りつぶされ(落とされ)ます。

**DROPPING LAND** → p. 396

## まとめ Stage04

本章では、変化に富んだステージを使っていろいろなアクションを行う「地形利用」について解説しました。地形利用を面白くするには、キャラクターのアクションと調和するような地形のデザインが重要です。例えば壁やロープを使ったアクションをとり入れるのならば、起伏が多い地形にした方がゲームが面白くなります。

というわけで、「キャラクターのアクションを生かせるように、地形をデザインしよう！」というのが本章のまとめです。

アクションゲームのキャラクターは、走ったりジャンプしたりするだけではありません。乗り物に乗ったり、仕掛けにつかまったりと、いろいろと特殊なアクションを行います。しゃがんだり、丸まったり、巨大化したりと、キャラクター自身の姿を変えるアクションもあります。

# 特殊行動

## Special Motion

ActionGame Algorithm Maniax

Stage

## スケートボード

キャラクターがスケートボードに乗って移動するアクションです。スケートボードに乗ると、普通に歩くよりも速いスピードで移動することができます。

スケートボードはFig. 5-1のような乗り物です。乗る方法はゲームによって違いますが、例えばスケートボードに接触してボタンを押すことで、乗ることができます(Fig. 5-2)。

スケートボードに乗っているときにレバーを操作すると、入力した方向にスケートボードが移動します(Fig. 5-3)。ボタンを押すと、スケートボードから降ります。

スケートボードを採用したゲームには、例えば「メトロクロス」があります。このゲームでも、スケートボードに乗ると普通に走るよりも速いスピードで移動することができます。また、走っているときにはスピードが落ちてしまうタイプの床の上を通っても、スケートボードに乗っていればスピードが落ちません。ただし、スケートボードに乗っている間はジャンプができないので、レバー操作だけで障害物を避ける必要があります。

Fig. 5-1 スケートボード

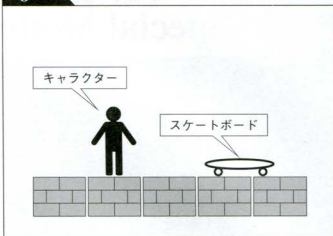
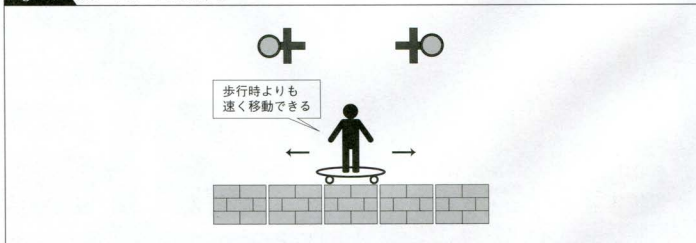



Fig. 5-2 スケートボードに乗る

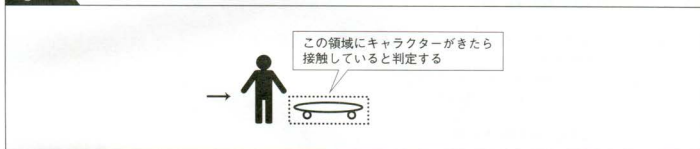
Fig. 5-3 スケートボードで移動する

「ワンダーボーイ」にもスケートボードがあります。このゲームではスケートボードに乗っていてもジャンプができます。また、敵に当たったときにはスケートボードから降ろされるだけで、ミスにならずにすみます。

## アルゴリズム Algorithm

スケートボードを実現するには、キャラクターがスケートボードに乗っている状態と、乗っていない状態を区別します。それには、スケートボードの形状に合わせた当たり判定を用意し、キャラクターとの間で当たり判定処理を行います(Fig. 5-4)。

キャラクターとスケートボードが接触しているときにボタンが押されたら、スケートボードに乗っている状態に移行します。スケートボードに乗っている状態でボタンが押されたら、再び乗っていない状態に戻します。

そして、スケートボードに乗っているときには、移動のスピードを速くします。ジャンプができないようにしたり、急な方向転換ができなくしても面白いでしょう。

**Fig. 5-4** スケートボードの当たり判定

## プログラム Program

List 5-1はスケートボードのプログラムです。処理を少し追加すれば、スケートボードに乗っているときにはジャンプをさせないとか、スピードが遅くならないといったルールを実現することもできます。

**List 5-1** スケートボード(CSkateboardManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 歩行の移動スピード
    float speed=0.1f;

    // スケートボードの移動スピード
    // 歩行時のスピードに対する倍率
    float skate_speed=3;
```

```
// スケートボードとの当たり判定処理を行うための定数
// X座標の差分の最大値
float max_dist=1.0f;

// レバーの入力に応じて左右に移動する
VX=0;
if (is->Left) VX=-speed;
if (is->Right) VX=speed;

// スケートボードに乗っているときには、移動スピードを速くする
// Skateboardはスケートボードへのポインタ
// スケートボードに乗っていると、SkateboardがNULL以外になる
if (Skateboard) VX*=skate_speed;

// X座標を更新し、キャラクターが画面からはみ出さないように補正する
X+=VX;
if (X<0) X=0;
if (X>MAX_X-1) X=MAX_X-1;

// ボタンを押したときの処理
if (!PrevButton && is->Button[0]) {

    // スケートボードに乗っていたら、スケートボードから降りる
    if (Skateboard) {
        Skateboard=NULL;
    } else

    // まだスケートボードに乗っておらず、
    // かつスケートボードに接触していたら、
    // スケートボードに乗る
    {
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (
                mover->Type==1 &&
                abs(X-mover->X)<max_dist
            ) {
                Skateboard=(CSkateboard*)mover;
            }
        }
    }
}

// スケートボードに乗っていたら、
// スケートボードがキャラクターと一体化して動くように、
// 両者のX座標を一致させる
// また、キャラクターがスケートボードに乗っているように見せるため、
// キャラクターのY座標を調整する
if (Skateboard) {
    Skateboard->X=X;
```

```
        Y=Skateboard->Y+0.05f;
    } else

    // スケートボードに乗っていないときには、
    // キャラクターが床面に立つようにY座標を調整する
    {
        Y=MAX_Y-2;
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // X方向の速度に応じて、キャラクターを傾けて表示する
    // スケートボードに乗っているときには傾けない
    Angle=Skateboard?0:VX/speed*0.05f;

    return true;
}
```

SAMPLE

「SKATEBOARD」はスケートボードのサンプルです。レバー入力でキャラクターが左右に移動します。スケートボードに接触している状態でボタンを押すと、スケートボードに乗ることができます。スケートボードに乗っている間は、通常よりも速く移動することができます。

**SKATEBOARD** → p. 396

# 自動車

動いている自動車の屋根に飛び乗るアクションです。キャラクターが自動車の屋根にうまく飛び乗ると、自動車に乗って移動することができます。

自動車はFig. 5-5のように地面を走っています。自動車に乗るには、自動車の近くでキャラクターをジャンプさせます (Fig. 5-6)。

うまく自動車の屋根に着地すれば、自動車に乗ることができます (Fig. 5-7)。キャラクターが乗っても、自動車は以前と変わらないスピードで走り続けます。乗っているキャラクターも、自動車に運ばれて同じスピードで進みます (Fig. 5-8)。

自動車から降りるには、ボタンを押して自動車からジャンプします (Fig. 5-9)。あるいは、レバー操作で左右に移動して、屋根から落ちてもかまいません。

自動車に乗るアクションを採用したゲームには「パックランド」があります。このゲームでは、モンスターが自動車を運転しながら迫ってきます。自動車にひかれるとミスになってしまうの

ですが、うまくジャンプして自動車の屋根に飛び乗れば、自動車に乗って進むことができます。あるいは、自動車の屋根を踏み台にして、高いところまでジャンプすることも可能です。

**Fig. 5-5** 自動車

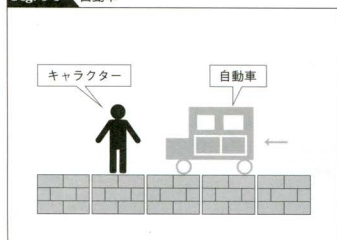
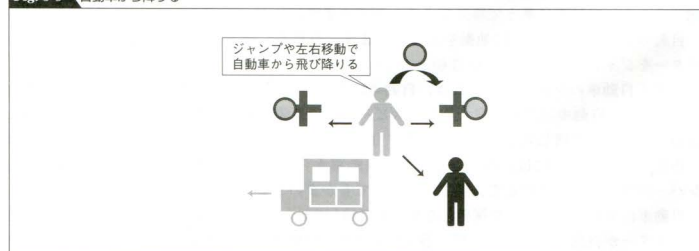



**Fig. 5-6** 自動車に乗るためにジャンプする

**Fig. 5-7** 自動車に飛び乗る

**Fig. 5-8** 自動車といっしょに進む

**Fig. 5-9** 自動車から降りる

## アルゴリズム Algorithm

自動車に乗るアクションを実現するには、自動車に乗ったかどうかの判定処理が必要です。落下中のキャラクターがFig. 5-10のような当たり判定のなかに入ったら、自動車に乗ったと判定します。キャラクターが落下中かどうかを調べるのは、自動車からジャンプして降りるときに、再び自動車に乗ったと誤って判定しないためです。

自動車に乗ったら、キャラクターを自動車と同じスピードで移動させます(Fig. 5-11)。自動車の速度をcvx、キャラクターの速度をVX、キャラクターのX座標をXとすると、

```
X+=cvx+VX
```

のように、X座標に対して自動車の速度とキャラクター自身の速度を加算します。両方の速度を加算することで、自動車に運ばれつつ、レバー操作で屋根の上を左右に移動するといったことが可能になります。

**Fig. 5-10** 自動車に乗ったかどうかの判定処理

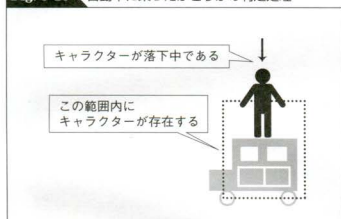



**Fig. 5-11** 自動車とキャラクターの速度

## プログラム Program

List 5-2は自動車に乗るアクションのプログラムです。自動車の当たり判定は、自動車の屋根の形に合わせてあります。同じように、ボンネットの形に合わせた当たり判定も用意すれば、キャラクターをボンネットに乗せることもできます。

**List 5-2** 自動車(CCarManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;
```

```
    // ジャンプの初速度
    float jump_speed=-0.4f;

    // ジャンプ中の加速度
    float jump_accel=0.02f;

    // 自動車との当たり判定処理を行うための定数
    // X座標とY座標の差分の最小値と最大値
    float min_x=-1.0f;
    float max_x=0.5f;
    float min_y=0.6f;
    float max_y=1.6f;

    // キャラクターが自動車に乗ったときに、
    // キャラクターのY座標を補正するための定数
    float adjust_y=1.1f;

    // ジャンプしていないときの処理
    if (!Jump) {

        // レバーの入力に応じて左右に移動する
        VX=0;
        if (is->Left) VX=-speed;
        if (is->Right) VX=speed;

        // ボタンを押したらジャンプする
        // Y方向の速度にジャンプの初速度を設定する
        if (is->Button[0]) VY=jump_speed;
    }

// X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // Y方向の速度を更新し、落下スピードが一定値を超えないように補正する
    VY+=jump_accel;
    if (VY>-jump_speed) VY=-jump_speed;

    // Y座標の更新
    Y+=VY;

    // ジャンプしているかどうかを調べる
    // Jumpはジャンプしているときにtrueにする変数
    Jump=true;

    // 地面にいるときにはジャンプしていない状態にする
    if (Y>MAX_Y-2) {
        Y=MAX_Y-2;
        Jump=false;
```

```
    }

    // 自動車に乗ったかどうかの判定処理
    // キャラクターが落下していて、
    // かつ自動車に接触していたら、乗ったと判定する
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            VY>0 &&
            mover->Type==1 &&
            mover->X-X>min_x &&
            mover->X-X<max_x &&
            mover->Y-Y>min_y &&
            mover->Y-Y<max_y
        ) {
            // 自動車に乗ったように見えるよう、
            // キャラクターの座標を調整する
            CCar* car=(CCar*)mover;
            X+=car->VX;
            Y=car->Y-adjust_y;

            // Y方向の速度を0にする
            VY=0;

            // ジャンプしていない状態にする
            Jump=false;
            break;
        }
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「CAR」は自動車に乗るアクションのサンプルです。レバーでキャラクターを左右に動かし、ボタンでジャンプさせられます。タイミングよくジャンプすることで、自動車の上に飛び乗ることができます。

**CAR** → p. 396

# 動物

キャラクターが動物に乗るアクションです。動物に乗って移動したり、敵を攻撃したりできます。動物に乗っているときには、歩いているときとは異なる強力な技が出たりします。

動物に乗るには、まず動物に近づきます(Fig. 5-12)。そして、動物に接触した状態でボタンを押します(Fig. 5-13)。動物に乗ると、キャラクターが動物にまたがるなど、乗っている様子を表すグラフィックになります(Fig. 5-14)。

動物に乗った状態でレバーを入力すると、キャラクターが自分で歩くときと同じように、レバーの入力方向へ移動することができます(Fig. 5-15)。ゲームによっては、動物に乗った状態では移動が速くなったり、特別な攻撃ができたりする場合もあります。

動物から降りるにはボタンを押します。ゲームによっては、動物に乗っているときに敵の攻撃を受けると、強制的に動物から降ろされる場合もあります。

動物を採用したゲームには、例えば「ゴールデンアックス」があります。このゲームでは動物に乗ると、動物を使った強力な攻撃ができます。ただし、動物に乗っている状態で攻撃を受けると、動物から降ろされてしまいます。動物に乗っている敵を攻撃して動物を奪ったり、逆に攻撃されて動物を奪われたりといったアクションも楽しめます。

動物に似たフィーチャーとして、戦車に乗るアクションを採用したゲームもあります。例えば「フロントライン」では、通常の主人公は歩行していますが、ステージに停めてある戦車や装甲車に乗ることができます。戦車に乗ると、攻撃を受けても一撃ではミスにならなくなります。攻撃を受けた戦車からは煙が出るので、爆発する前に降りて脱出する必要があります。壊れかけの戦車をぎりぎりまで使うのも、このゲームの面白い遊び方の1つです。

戦車に乗れるゲームには、ほかにも「メタルスラッグ」などがあります。また「ラッシュ&クラッシュ」では、自動車に乗ることができます。自動車に乗ったときには、歩いているときとは違って、レバー入力でハンドルを回すような操作方法になります。

**Fig. 5-12** 動物

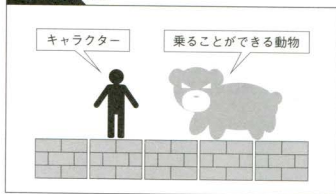



**Fig. 5-13** 動物に乗る

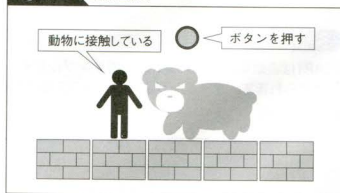
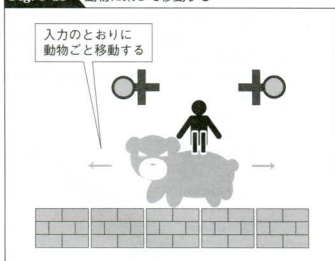



**Fig. 5-14** 動物に乗った状態

**Fig. 5-15** 動物に乗って移動する

## アルゴリズム Algorithm

動物に乗るアクションを実現する方法は、実は「スケートボード（→ p. 268）」の実現方法と同じです。動物に接触しているときにボタンを押したら動物に乗り、動物に乗っているときにボタンを押したら降ります。動物に乗っていたら、歩行時とは移動速度や攻撃方法を変えます。

## プログラム Program

List 5-3は動物に乗るアクションのプログラムです。このサンプルでは、動物に乗っているときには移動スピードが速くなります。少し処理を追加すれば、動物に乗っているときに攻撃を強くしたり、特別な技が出るようにしたりといったこともできます。

**List 5-3** 動物（CAnimalManクラス）

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.1f;

    // 動物に乗ったときの移動スピード
    // 通常の移動スピードに対する比率
    float animal_speed=3;

    // 動物との当たり判定処理を行うための定数
    // X座標の差分の最大値
    float max_dist=1.0f;
```

**List 5-3**

```
// レバーの入力に応じて左右に移動する
// 動物の向きを決めるために、
// キャラクターが移動した方向を保存しておく
// VXとVYはキャラクターの速度を表す変数
// DirXとDirYはキャラクターの移動方向を表す変数
VX=0;
if (is->Left) {
    DirX=-1;
    VX=-speed;
}
if (is->Right) {
    DirX=1;
    VX=speed;
}

// 動物に乗っているときには移動スピードを速くする
if (Animal) VX*=animal_speed;

// X座標を更新し、画面からはみ出さないように補正する
X+=VX;
if (X<0) X=0;
if (X>MAX_X-1) X=MAX_X-1;

// ボタンを押したときの処理
if (!PrevButton && is->Button[0]) {

    // 動物に乗っていたら、動物から降りる
    // Animalは動物へのポインタを保持する変数
    if (Animal) {
        Animal=NULL;
    } else

    // 動物に乗っていなかったら、
    // 動物に接触しているかどうかを調べる
    // 接触していたら、その動物に乗る
    {
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (
                mover->Type==1 &&
                abs(X-mover->X)<max_dist
            ) {
                Animal=(CAnimal*)mover;
                break;
            }
        }
    }
}

// 動物に乗っているときの処理
if (Animal) {
```

```
        // 座っているキャラクターの絵を表示する
        Texture=Game->Texture[TEX_CROUCH];

        // 動物のX座標をキャラクターのX座標に合わせる
        Animal->X=X+DirX*0.5f;

        // Y座標を調整して、キャラクターを動物の上に配置する
        Y=Animal->Y-0.5f;

        // 移動方向に応じて動物の向きを変える
        Animal->ReverseX=(DirX>0);
    } else

    // 動物に乗っていないときの処理
    {
        // 立っているキャラクターの絵を表示する
        Texture=Game->Texture[TEX_MAN];

        // Y座標を調整して、キャラクターを床の上に配置する
        Y=MAX_Y-2;
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=Animal?0:VX/speed*0.05f;

    return true;
}
```

SAMPLE

「ANIMAL」は動物に乗るアクションのサンプルです。レバーでキャラクターを左右に動かすことができます。動物に接触している状態でボタンを押すと、動物の上に乗ることができます。動物に乗っている間は、通常よりも速く移動することができます。

**ANIMAL** → p. 396

# シーソー

シーソーを使ってキャラクターが高くジャンプするアクションです。シーソーの片側にキャラクターが乗っているときに、反対側に別のキャラクターが飛び乗ると、乗っていたキャラクターが空中に跳ね上げられます。

ここではキャラクターではなく、シーソーを操作する場合を考えてみましょう(Fig. 5-16)。シーソーはレバー入力で左右に動きます。空からは人のキャラクターが降ってきます。

シーソーの左右には、それぞれキャラクターを1人ずつ乗せることができます。誰も乗っていない場合には、左右どちら側でも乗せることができます。片側にキャラクターが乗っている場合には、空いている反対側で降ってくるキャラクターを受け止める必要があります(Fig. 5-17)。

降ってくるキャラクターを受け止めるとシーソーは下がります(Fig. 5-18)。そして、反対側に乗っていたキャラクターは、反動で空中に跳ね上げられます。跳ね上げたキャラクターを再び空いている側で受け止め、反対側のキャラクターを跳ね上げ…という操作を繰り返すと、お

**Fig. 5-16** シーソー

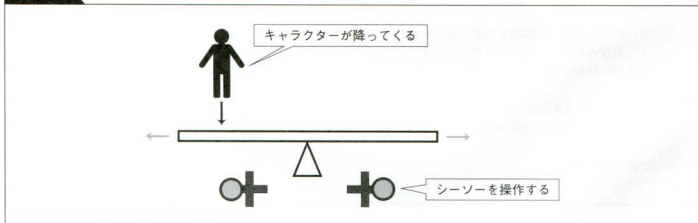



**Fig. 5-17** キャラクターを受け止める

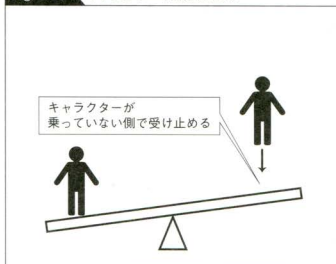
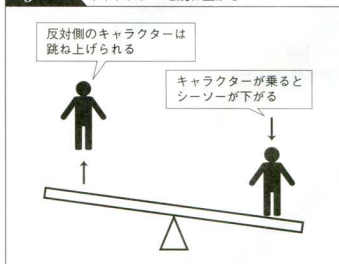



**Fig. 5-18** キャラクターを跳ね上げる

手玉のようにキャラクターを跳ね上げ続けることができます。

シーソーを採用したゲームには、例えば「サーカス」があります。このゲームではシーソーを操作して、降ってくる人を空いている側で受け止め、反対側に乗っていた人を跳ね上げます。そして、跳ね上げた人を使って、空中に浮かんでいるアイテムを回収します。

また、シーソーではなく人を操作するゲームもあります。例えば「ポパイ」では、キャラクターを操作してシーソーに飛び乗ると、高くジャンプすることができます。

## アルゴリズム Algorithm

シーソーを実現するには、キャラクターがシーソーに乗ったことを判定する必要があります(Fig. 5-19)。シーソーの当たり判定は、ほかのキャラクターが乗っていない側にあります。この当たり判定にキャラクターが接触して、かつキャラクターが落下中ならば、シーソーに乗ったと判定します。

キャラクターが乗ったら、乗った側を下げます。そして、反対側に乗っていたキャラクターを跳ね上げます。空になった側が、新たな当たり判定を持ちます(Fig. 5-20)。

**Fig. 5-19** シーソーにキャラクターを乗せる処理

キャラクターが
落下中である
この範囲に
キャラクターが存在する

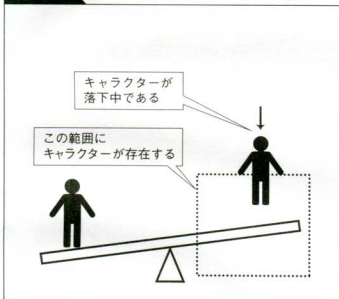
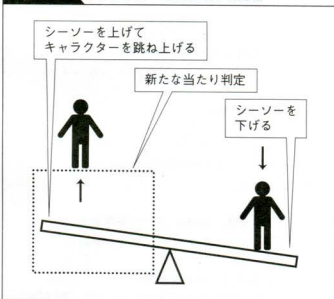



**Fig. 5-20** キャラクターを跳ね上げる処理

## プログラム Program

List 5-4はシーソーのプログラムです。このサンプルでは、空から降ってくるキャラクターをシーソーで受け止めます。シーソーの空いている側で受け止めると、反対側に乗っているキャラクターが跳ね上がります。左右交互に受け止めて、キャラクターをお手玉のように跳ね上げることが目的です。動きを面白くするため、飛び乗る位置によって、キャラクターを跳ね上げる角度が微妙に変わるようにしています。

**List 5-4** シーソー(CSeesawクラス、CSeesawManクラス)

```
// シーソーの移動処理を行うMove関数
bool CSeesaw::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 傾いたときの角度
    float max_angle=0.05f;

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、シーソーが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // キャラクターが乗っている側が下がるように、角度を調整する
    //
    Angle=0;
    if (Man[0]) Angle=-max_angle;
    if (Man[1]) Angle=max_angle;

    return true;
}

// シーソーの描画処理を行うDraw関数
void CSeesaw::Draw() {

    // X座標・Y座標・角度を保存する
    float x=X, y=Y, angle=Angle;

    // シーソーの土台を表示する
    Texture=Game->Texture[TEX_SEESAW0];
    Y=y+0.1f;
    W=H=1;
    Angle=0;
    CMover::Draw();

    // シーソーの板を表示する
    Texture=Game->Texture[TEX_SEESAW1];
    Y=y-0.3f;
    W=4;
    Angle=angle;
    CMover::Draw();
```

```
    // X座標・Y座標・角度を元の値に戻す
    X=x;
    Y=y;
    Angle=angle;
}

// キャラクターの移動処理を行うMove関数
bool CSeesawMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // ジャンプの初速度
    float jump_speed=-0.4f;

    // ジャンプ中の加速度
    float jump_accel=0.008f;

    // シーソーとの当たり判定処理を行うための定数
    // X座標の差分の最大値、Y座標の差分の最小値と最大値
    float max_x=2.0f;
    float min_y=0.5f;
    float max_y=1.5f;

    // シーソーに乗ったときに、
    // キャラクターの座標を調整するための定数
    float ride_x=1.5f;
    float ride_y=-0.3f;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<-1) X=MAX_X;
    if (X>MAX_X) X=-1;

    // Y方向の速度を更新し、落下スピードが一定値を超えないように補正する
    VY+=jump_accel;
    if (VY>-jump_speed) VY=-jump_speed;

    // Y座標を更新し、画面からはみ出したときには位置を初期化する
    Y+=VY;
    if (Y>MAX_Y) Init();

    // シーソーに乗ったかどうかの判定処理
    // 落下中にシーソーの当たり判定に接触したら、乗ったと判定する
    if (
        VY>0 &&
        abs(Seesaw->X-X)<max_x &&
        Seesaw->Y-Y>min_y &&
        Seesaw->Y-Y<max_y
    ) {
```

**List 5-4**

```
    // シーソーの左側にほかのキャラクターが乗っておらず、
    // かつシーソーの左側に接触したら、左側に乗る
    // シーソーのManメンバにキャラクターへのポインタを登録する
    if (!Seesaw->Man[0] && X<Seesaw->X) {
        Seesaw->Man[0]=this;

        // シーソーの右側にキャラクターが乗っていたら、
        // 空中に跳ね上げる
        if (Seesaw->Man[1]) {
            Seesaw->Man[1]->VX=speed*((Seesaw->X-X)/max_x-0.5f);
            Seesaw->Man[1]->VY=jump_speed;
            Seesaw->Man[1]=NULL;
        }
    }

    // シーソーの右側にほかのキャラクターが乗っておらず、
    // かつシーソーの右側に接触したら、右側に乗る
    // シーソーのManメンバにキャラクターへのポインタを登録する
    if (!Seesaw->Man[1] && X>Seesaw->X) {
        Seesaw->Man[1]=this;

        // シーソーの左側にキャラクターが乗っていたら、
        // 空中に跳ね上げる
        if (Seesaw->Man[0]) {
            Seesaw->Man[0]->VX=speed*((Seesaw->X-X)/max_x+0.5f);
            Seesaw->Man[0]->VY=jump_speed;
            Seesaw->Man[0]=NULL;
        }
    }
}

// シーソーの左側に乗っていたら、
// 左側にちょうど乗るように座標を調整する
if (Seesaw->Man[0]==this) {
    X=Seesaw->X-ride_x;
    Y=Seesaw->Y+ride_y;
}

// シーソーの右側に乗っていたら、
// 右側にちょうど乗るように座標を調整する
if (Seesaw->Man[1]==this) {
    X=Seesaw->X+ride_x;
    Y=Seesaw->Y+ride_y;
}

// X方向の速度に応じて、キャラクターを傾けて表示する
Angle=VX/speed*0.1f;

return true;
}
```

「SEESAW」はシーソーのサンプルです。レバーでシーソーを左右に移動させることができます。シーソーを操作して上から降ってくるキャラクターを受け止めます。シーソーの空いている側でキャラクターを受け止めると、反対側に乗っていたキャラクターが跳ね上がります。

**SEESAW** → p. 396

## 振り子

左右に揺れる振り子です。ジャンプして振り子につかまったり、振り子から振り子に飛び移ったりといったアクションが可能です。

振り子は左右に振動しています (Fig. 5-21)。ボタンを押してジャンプし、うまく振り子に飛び付くと (Fig. 5-22)、ぶら下がって振り子といっしょに揺れることができます (Fig. 5-23)。

振り子につかまった状態でボタンを押すと、ジャンプして振り子を離れることができます (Fig. 5-24)。さらに別の振り子にうまく飛び付くと、再び振り子につかまって揺れることができます。

振り子を採用したゲームには、例えば「サーカスチャーリー」があります。このゲームでは地面に落ちないように、振り子から振り子へジャンプしながら進んでいきます。単に振り子から振り子へ飛び移るだけではなく、振り子にぶら下がっている人につかまったり、トランポリンでジャンプしてから振り子につかまったりと、幅広いアクションが楽しめます。

**Fig. 5-21** 振り子

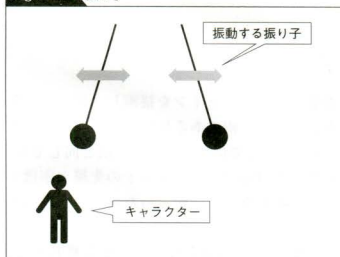



**Fig. 5-22** 振り子に飛びつく

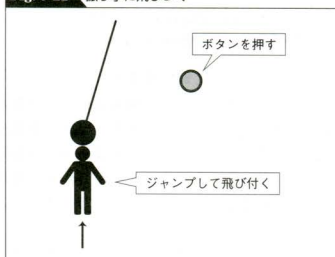
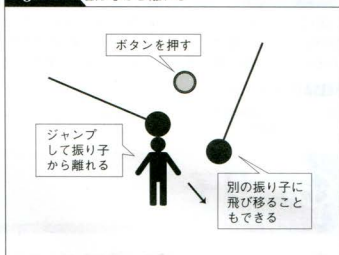



**Fig. 5-23** 振り子といっしょに揺れる

**Fig. 5-24** 振り子から離れる

「ピットフォール2」にも振り子が登場します。このゲームでは、左右に揺れるロープを使って池を跳び越えます。

なお、振り子と似た動きの仕掛けとしては、左右に揺れる鎌があります。こちらは振り子とは違い、キャラクターは鎌に当たらないように通り抜ける必要があります。動きとしては振り子と同じなので、キャラクターが接触したときの処理を変えれば、鎌を実現することができます。

## アルゴリズム Algorithm

振り子を実現するには、まず振り子を揺らす処理が必要です (Fig. 5-25)。振り子を揺らすには、振り子の角度を時間とともに変化させます。振り子の角度をAngle、根元の座標を (RootX, RootY)、振り子の長さをLengthとすると、振り子の先端の座標 (X, Y) は、

```
X=RootX+sinf(Angle)*Length
Y=RootY+cosf(Angle)*Length
```

のように求めることができます。

振り子を描画するには、例えば根元の座標から先端の座標までラインを描画し、振り子の先端には円形の画像を表示します。ラインを描くときには、少し幅のある矩形を描いた方が見やすいでしょう。矩形を描く方法は、「ロープを張る (→ p. 235)」でロープを描く方法と同じです。

次に、振り子にキャラクターがつかまる処理が必要です (Fig. 5-26)。振り子の先端の座標を求めて、先端付近に当たり判定を設けます。この範囲にキャラクターが入ったら、つかまったと判定します。

振り子から離れる処理にも注意が必要です (Fig. 5-27)。ジャンプして振り子から離れたら、その振り子から一度完全に離れるまで、再び同じ振り子にはつかまらないようにします。こうしないと、つかまっていた振り子から離れた瞬間に、また同じ振り子に接触してつかまってしまい、振り子から離れられなくなる可能性があります。

**Fig. 5-25** 振り子を揺らす処理

**Fig. 5-26** 振り子につかまる処理

この範囲にキャラクターがいたら
振り子につかまったと判定する

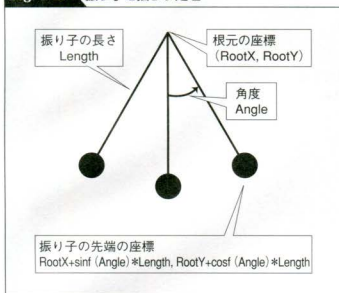
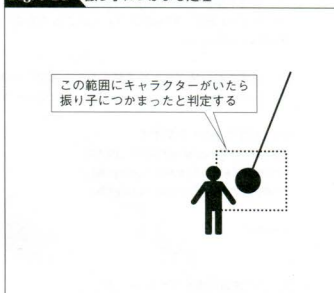
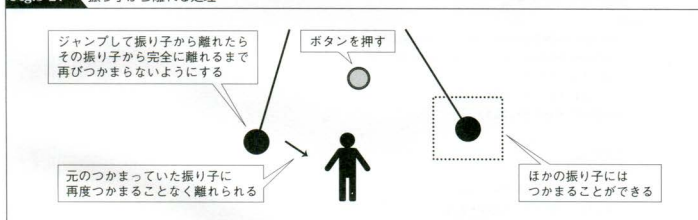



**Fig.5-27** 振り子から離れる処理

## プログラム Program

List 5-5は振り子のプログラムです。このサンプルでは振り子を一定の速度で揺らしていますが、加速度を使うと動きがよりリアルになります。加速度を使う場合は、振り子が真下に向いているときにはスピードを速く、左右の端では遅くするとよいでしょう。

**List 5-5** 振り子(CPendulumクラス、CPendulumManクラス)

```
// 振り子の移動処理を行うMove関数
bool CPendulum::Move(const CInputState* is) {

    // 角度の最大値
    float max_angle=0.2f;
```

**List 5-5**

```
    // 角度を更新する
    // Angleは角度、VAngleはフレームごとの角度の変化
    Angle+=VAngle;

    // 角度の絶対値が最大値を超えたら、速度の向きを逆にする
    if (abs(Angle)>=max_angle) VAngle=-VAngle;

    // 先端の座標を計算する
    float rad=Angle*D3DX_PI*2;
    X=RootX+sinf(rad)*Length;
    Y=RootY+cosf(rad)*Length;

    return true;
}

// 振り子の描画処理を行うDraw関数
void CPendulum::Draw() {

    // 座標計算のための準備
    float
        w=Game->GetGraphics()->GetWidth()/MAX_X,
        h=Game->GetGraphics()->GetHeight()/MAX_Y,
        rad=Angle*D3DX_PI*2,
        c=cosf(rad)*0.05f,
        s=sinf(rad)*0.05f;

    // 矩形のX座標
    float x[]={
        (RootX-c+0.5f)*w, (RootX+c+0.5f)*w,
        (X-c+0.5f)*w, (X+c+0.5f)*w
    };

    // 矩形のY座標
    float y[]={
        (RootY+s+0.5f)*h, (RootY-s+0.5f)*h,
        (Y+s+0.5f)*h, (Y-s+0.5f)*h
    };

    // 振り子の矩形部分を描画する
    Game->Texture[TEX_FILL]->Draw(
        x[0], y[0], Color, 0, 0,
        x[1], y[1], Color, 0, 0,
        x[2], y[2], Color, 0, 0,
        x[3], y[3], Color, 0, 0
    );

    // 振り子の先端に円形の画像を表示する
    Texture=Game->Texture[TEX_BALL];
    CMover::Draw();
}
```

```
// キャラクターの移動処理を行うMove関数
bool CPendulumMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // ジャンプの初速度
    float jump_speed=-0.4f;

    // ジャンプ中の加速度
    float jump_accel=0.02f;

    // 振り子との当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=1.0f;

    // ジャンプしていないときの処理
    if (!Jump) {

        // レバーの入力に応じて左右に移動する
        VX=0;
        if (is->Left) VX=-speed;
        if (is->Right) VX=speed;

        // ボタンを押したらジャンプする
        if (!PrevButton && is->Button[0]) {
            VY=jump_speed;
            Pendulum=NULL;
            Jump=true;
        }
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // X座標を更新し、画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // Y方向の速度を更新し、落下スピードが一定値を超えないように補正する
    VY+=jump_accel;
    if (VY>-jump_speed) VY=-jump_speed;

    // Y座標の更新
    Y+=VY;

    // 床に接触したら、ジャンプ状態を解除する
    if (Y>=MAX_Y-2) {
```

**List 5-5**

```
		Y=MAX_Y-2;
		Jump=false;
	}

	// 直前につかまっていた振り子から離れたかどうかの判定処理
	// PrevPendulumは直前につかまっていた振り子へのポインタ
	// 振り子と接触しなくなったら、ポインタをNULLにする
	if (
		PrevPendulum &&
		(abs(PrevPendulum->X-X)>=max_dist ||
		abs(PrevPendulum->Y-Y)>=max_dist)
	) {
		PrevPendulum=NULL;
	}

	// 振り子につかまっているときの処理
	// キャラクターの座標を振り子の先端の座標に合わせる
	// Pendulumは現在つかまっている振り子へのポインタ
	if (Pendulum) {
		X=Pendulum->X;
		Y=Pendulum->Y;
	} else

	// 振り子につかまっていないときの処理
	// 振り子に接触したら、振り子につかまる
	// ただし、直前につかまっていた振り子にはつかまらない
	// 直前につかまっていた振り子から一度完全に離れたら、
	// 再びつかまることができる
	{
		for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
			CMover* mover=(CMover*)i.Next();
			if (
				mover->Type==1 &&
				abs(mover->X-X)<max_dist &&
				abs(mover->Y-Y)<max_dist &&
				mover!=PrevPendulum
			) {
				// つかまった振り子を記録し、ジャンプ状態を解除する
				Pendulum=PrevPendulum=(CPendulum*)mover;
				Jump=false;
				break;
			}
		}
	}

	// X方向の速度に応じて、キャラクターを傾けて表示する
	Angle=VX/speed*0.1f;

	return true;
}
```

SAMPLE

「PENDULUM」は振り子のアクションのサンプルです。レバーでキャラクターが左右に移動します。振り子に向かってタイミングよくジャンプすると、振り子につかまることができます。振り子につかまった状態から再度ジャンプして、別の振り子に飛び移ることもできます。

**PENDULUM** → p. 397

## しゃがむ

キャラクターがしゃがむアクションです。しゃがんだまま歩く「しゃがみ歩き」のアクションを採用したゲームもあります。

通常状態のキャラクターは立っています(Fig. 5-28)。レバーの下に入れると、キャラクターはしゃがみます(Fig. 5-29)。

レバーを左下や右下に入れると、キャラクターはしゃがんだまま歩きます(Fig. 5-30)。しゃがみ歩きのスピードは、立って歩くときのスピードに比べると遅いのですが、敵の攻撃を避けたり、狭いところをくぐり抜けたりするときには便利です。

**Fig. 5-28** 立っているキャラクター

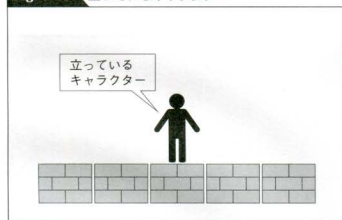
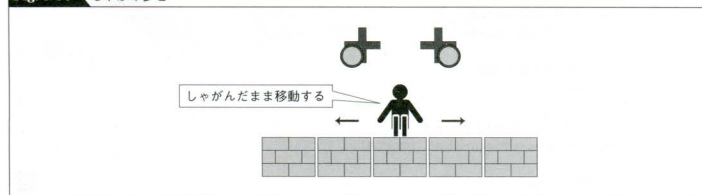



**Fig. 5-29** しゃがんでいるキャラクター

**Fig. 5-30** しゃがみ歩き

しゃがむアクションを採用したゲームは数多くあります。しゃがみ歩きも含めて採用したゲームには、例えば「ニンジャウォリアーズ」があります。しゃがみ歩きは通常の歩行に比べて少し遅いのですが、銃弾などを避けながら敵を攻撃することができます。

同じくしゃがみ歩きを採用したゲームには、「忍」や「シャドウダンサー」もあります。このように忍者の主人公のゲームでは、しゃがみ歩きを採用することが多いようです。

## アルゴリズム Algorithm

しゃがみとしゃがみ歩きの実現方法は簡単です。レバーが下方向に入っていたら、キャラクターをしゃがませます。しゃがんだ様子を表すグラフィックを用意して、表示を切り替えましょう。

レバーが斜め下方向に入っているときには、通常の歩行よりも遅いスピードでキャラクターを左右に移動させます。これも、レバーの方向を取得すれば簡単に処理できます。

## プログラム Program

List 5-6はしゃがむアクションのプログラムです。しゃがんだときには、移動スピードを立っているときの30%に落としています。

**List 5-6** しゃがむ(CCrouchManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // しゃがみ歩きの移動スピード
    // 通常の移動スピードに対する比率
    float crouch_speed=0.3f;

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // しゃがんでいるときの処理
    if (is->Down) {

        // しゃがんでいる画像を表示する
        Texture=Game->Texture[TEX_CROUCH];

        // 移動スピードを遅くする
```

```
        VX*=crouch_speed;

        // 画像の大きさや表示位置の調整
        W=H=0.8f;
        Y=MAX_Y-1.8f;
    } else

    // 立っているときの処理
    {
        // 立っている画像を表示する
        Texture=Game->Texture[TEX_MAN];

        // 画像の大きさや表示位置の調整
        W=H=1;
        Y=MAX_Y-2;
    }

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

## SAMPLE

「CROUCH」はしゃがみのアクションのサンプルです。レバーでキャラクターが左右に移動します。レバーを下に入れるとキャラクターはしゃがみます。レバーを斜め下方向に入れれば、しゃがんだ状態で左右に移動することもできます。

**CROUCH** → p. 397

# 丸まる

キャラクターが丸くなって、地面を転がるアクションです。狭いところに入ったり、通常よりも速く移動したりすることができます。

レバーを下方向に入れると、キャラクターが丸まります。斜め下方向に入れた場合には、丸まったまま転がって左右に移動します（Fig. 5-31）。レバーを下に入れるのをやめると、キャラクターは立ち上がって止まります（Fig. 5-32）。丸まっているときに移動方向を変えるには、一度立ち上がる必要があります。

丸まるアクションを採用したゲームには、例えば「メトロイド」があります。丸まることによって、敵の攻撃をかわしやすくなったり、狭いところに入れたりします。また、丸まった状態から出す特別な攻撃もあります。

「ソニック・ザ・ヘッジホッグ」も丸まるアクションを採用したゲームです。このゲームでは丸くなって転がることによって、移動スピードを上げることができます。ステージには坂道やループなど、転がって抜けると楽しい仕掛けが数多く用意されています。

**Fig. 5-31** 丸まって移動する

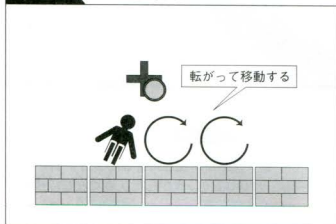
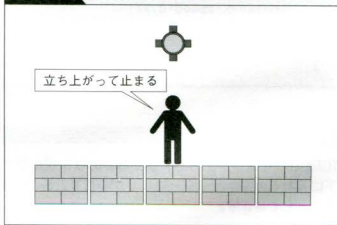



**Fig. 5-32** 立ち上がって止まる

## アルゴリズム Algorithm

丸まるアクションを実現する方法は、しゃがみ歩きの実現方法とほとんど同じです。レバーが下方向に入っていたら、キャラクターを丸まらせます。レバーが斜め下方向に入っているときには、その方向にキャラクターを転がします。レバーを中央に戻したら、移動をやめて、キャラクターを立ち上がらせます。

## プログラム Program

List 5-7は丸まるアクションのプログラムです。丸まっているときの処理を追加すれば、丸まった状態から出す技なども実現することもできます。

**List 5-7** 丸まる(CCurlUpManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 回転のスピード
    float vangle=0.05f;

    // 丸まっていないときの処理
    if (!CurlUp) {

        // レバーの入力に応じて左右に移動する
        VX=0;
        if (is->Left) VX=-speed;
        if (is->Right) VX=speed;

        // レバーを下に入れたらしゃがむ
        if (is->Down) CurlUp=true;

        // 立っている画像を表示する
        Texture=Game->Texture[TEX_MAN];

        // 画像の大きさ・表示位置・角度の調整
        W=H=1;
        Y=MAX_Y-2;
        Angle=VX/speed*0.1f;
    } else

    // 丸まっているときの処理
    {
        // レバーを下に入れていなければ立ち上がる
        if (!is->Down) CurlUp=false;

        // 丸まっている画像を表示する
        Texture=Game->Texture[TEX_CROUCH];

        // 画像の大きさ・表示位置・角度の調整
        W=H=0.8f;
        Y=MAX_Y-1.8f;
        Angle+=vangle*VX/speed;
```

**List 5-7**

```
    }

    // X座標を更新し、画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    return true;
}
```

SAMPLE

「CURLING」は丸まるアクションのサンプルです。レバーでキャラクターが左右に移動します。レバーを下に入れるとキャラクターは丸まります。レバーを斜め下に入れると、キャラクターは丸まったまま高速で移動します。

**CURLING** → p. 397

## 巨大化

キャラクターの体が大きくなって、移動速度や攻撃力がアップするアクションです。巨大化するための方法はゲームによって違います。アイテムを取ったときに巨大化するゲームもあれば、ゲージがあるときにボタンを押すことで巨大化できるゲームもあります。

ここでは単純に、ボタンを押したときに巨大化する例を説明します（Fig. 5-33）。通常状態でボタンを押すと、キャラクターがしだいに大きくなって、巨大化します（Fig. 5-34）。巨大化状態でボタンを押すと、だんだん小さくなって通常状態に戻ります。

巨大化を採用したゲームには、例えば「スーパーマリオブラザーズ」があります。このゲームでは特定のキノコを拾うと、主人公が巨大化します。巨大化状態では体当たりで壁を壊すことができます。また、巨大化状態で敵に触れたときには、すぐにミスにはならず、小さくなるだけですみます。

巨大化するのではなく、変身するゲームもあります。例えば「獣王記」では、パワーアップアイテムを取るたびにキャラクターの筋肉が増えていき、最後には強力な攻撃ができる獣に変身します。

**Fig. 5-33** 通常状態

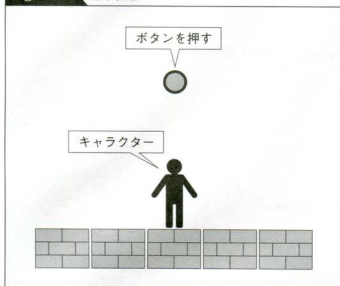
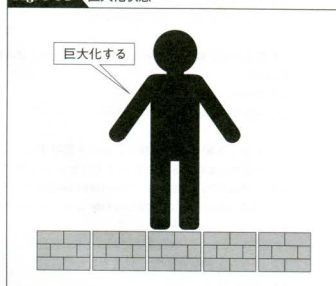



**Fig. 5-34** 巨大化状態

## アルゴリズム Algorithm

巨大化を実現するには、通常状態と巨大化状態を区別します。通常状態でボタンを押したり、アイテムを取ったりしたら、巨大化状態に移行します。巨大化状態では、通常よりも移動スピードや攻撃力をアップさせます。巨大化状態でボタンを押したり、敵の攻撃を受けたりしたら、通常状態に戻ります。

## プログラム Program

List 5-8は巨大化のプログラムです。このサンプルでは巨大化時に移動スピードを速くしています。少し処理を追加すれば、巨大化時に攻撃力が強くなったり、特別な攻撃が出たりといったことも実現できます。また、通常状態と巨大化状態の間を移行するときには、キャラクターの大きさが滑らかに変化するようにしています。

**List 5-8** 巨大化(CGiantManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.1f;

    // 体の大きさが変化するスピード
    float vsize=0.05f;

    // レバーの入力に応じて左右に移動する
    VX=0;
```

**List 5-8**

```
    if (is->Left) VX=-speed*W;
    if (is->Right) VX=speed*W;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // ボタンを押したら、状態の移行を開始する
    // VSizeは体の大きさが変化するスピードを表す
    if (!PrevButton && is->Button[0]) {
        if (Size==1) VSize=vsize; else VSize=-vsize;
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // 体の大きさを変化させる
    Size+=VSize;

    // 体の大きさに応じて、表示のサイズや位置を調整する
    W=H=Size;
    Y-=VSize/2;

    // 体が一定サイズ以下には小さくならないようにするための処理
    if (Size<=1) {
        Size=W=H=1;
        Y=MAX_Y-2;
    }

    // 体が一定サイズ以上には大きくならないようにするための処理
    if (Size>=4) {
        Size=W=H=4;
        Y=MAX_Y-3.5f;
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed/W*0.1f;

    return true;
}
```

SAMPLE

「GIANT」は巨大化のアクションのサンプルです。レバーでキャラクターが左右に移動し、ボタンを押すと巨大化します。巨大化中は、通常よりも速い速度で移動することができます。巨大化中にボタンを押すと、元の大きさに戻ります。

**GIANT** → p. 397

# 複数キャラクターの操作

1人のプレイヤーが複数のキャラクターを同時に操作するアクションです。ゲームによっては、複数人のプレイヤーが複数のキャラクターを手分けして動かせる場合もあります。

AとBという複数のキャラクターがいる例を考えましょう(Fig. 5-35)。1人のプレイヤーが複数のキャラクターを動かすやり方としては、例えば次のような方法が考えられます。

・ボタンなどで動かすキャラクターを切り替える
・キャラクターの数だけレバーを用意する
・片方のキャラクターは自動的に動く

比較的実現しやすいのは、動かすキャラクターをボタンで切り替える方法です。例えば、片方のキャラクターはレバー操作で動かし、もう片方のキャラクターはボタンを押しながらレバーを操作することによって動かす、といった方法があります(Fig. 5-36)。

**Fig. 5-35** 複数のキャラクター

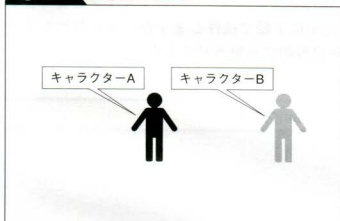
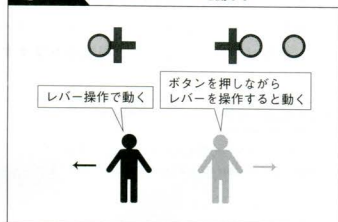



**Fig. 5-36** 複数のキャラクターを動かす

複数キャラクターの操作を採用したゲームには、例えば「イシターの復活」があります。このゲームには2体のキャラクターがいて、それぞれのキャラクターを動かすために2本のレバーが用意されています。1人のプレイヤーが2体のキャラクターを動かすことも、2人のプレイヤーが1体ずつ手分けして動かすこともできます。ただし、片方のキャラクターは基本的に敵に体当たりするだけなので、このキャラクターを担当するプレイヤーは少し退屈するかもしれません。そういった意味では、1人で2体のキャラクターを動かしてプレイすることをかなり意識したゲームだといえます。

1人のプレイヤーが複数のキャラクターを動かすゲームは少ないのですが、複数のプレイヤーが同時に遊べるゲームは数多くあります。例えば「ワルキューレの伝説」は2人で、「ガントレット」は4人で遊ぶことができます。これらのゲームのキャラクターは、複数のプレイヤーが遊ぶことを想定しているため、1人のプレイヤーが複数のキャラクターを動かすのは、不可能ではありませんが困難です。

またMMORPGですが、「グラナド・エスパダ」は1人で3人のキャラクターを同時に操作するゲームです。キャラクターは入力したコマンドにしたがいつつも、索敵や攻撃などをある程度自動的に行います。そのため、かなりアクション性が高いゲームながらも、スムーズに複数のキャラクターを同時に動かすことができます。キャラクターの動きを半自動化しているという点では、RTS(リアルタイムストラテジー)ゲームに近い操作性だといえます。

## アルゴリズム Algorithm

ボタンを押しているかどうかで動かすキャラクターを切り替える方法の場合には、複数キャラクターの操作を実現するのは簡単です。ボタンを押しているときにはキャラクターAを、押していないときにはキャラクターBを、それぞれレバー操作に応じて動かすだけです。

## プログラム Program

List 5-9は複数キャラクターの操作を行うプログラムです。キャラクターAとキャラクターBは、同一の処理で動かします。

複数のキャラクターを1人のプレイヤーに同時に操作させるには、ゲームデザインをよく練る必要があります。このサンプルはキャラクターを完全に手動で操作しますが、より複雑なアクションをさせる場合には、ある程度キャラクターを自動的に行動させる必要があるでしょう。

**List 5-9** 複数キャラクターの操作(CMultipleCharacterManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // キャラクターAはボタンを押しているときに動かす
    // キャラクターBはボタンを押していないときに動かす
    // UseButtonはキャラクターを動かすときのボタンの状態を表す
    // キャラクターによって、UseButtonをtrueまたはfalseに設定しておく
    if (is->Button[0]==UseButton) {

        // レバーの入力に応じて左右上下に移動する
        VX=VY=0;
        if (is->Left) VX=-speed; else
        if (is->Right) VX=speed; else
        if (is->Up) VY=-speed; else
        if (is->Down) VY=speed;

        // X座標を更新し、キャラクターが画面からはみ出さないように補正する
        X+=VX;
```

```
        if (X<0) X=0;
        if (X>MAX_X-1) X=MAX_X-1;

        // Y座標を更新し、キャラクターが画面からはみ出さないように補正する
        Y+=VY;
        if (Y<0) Y=0;
        if (Y>MAX_Y-1) Y=MAX_Y-1;
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「MULTIPLE CHARACTER」は複数キャラクターの操作のサンプルです。レバーでキャラクターが上下左右に移動します。ボタンを押している間は、もう1人のキャラクターを操作することができます。

**MULTIPLE CHARACTER** → p. 397

## まとめ Stage05

本章では、キャラクターがいろいろと変わったアクションを行う「特殊行動」について解説しました。特殊行動はゲームを強く特徴づける要素です。数多くの特殊行動を取り入れるのも悪くはないのですが、1つの特殊行動について掘り下げることで、プレイヤーの印象に残るゲームを作ることができるでしょう。

というわけで、「奥深い特殊行動を取り入れて、ゲームの印象を強くしよう！」というのが本章のまとめです。

アクションゲームで敵を攻撃する方法は、パンチやキックだけではありません。ゲームによっては、いろいろな武器を使った攻撃が可能です。剣やムチといった近接攻撃もあれば、手榴弾やマシンガンといった遠隔攻撃もあります。また、誘導ミサイルやブーメランのような、特殊な動きをする武器もあります。

# 武器
## Weapon

ActionGame Algorithm Maniax

Stage

# 剣

剣を振って敵を斬る攻撃です。剣の長さはゲームによって違いますが、キャラクターの比較的近くにいる敵を攻撃する点は共通しています。

ボタンを押すと、剣を振ることができます (Fig. 6-1)。剣を振っている間に、うまく敵に剣を当てると、敵を斬ってダメージを与えられます (Fig. 6-2)。

剣を振る様子の表現方法は、ゲームによって異なります。シンプルに剣を突き出すだけのゲームもあれば、剣の画像を回転させたり、アニメーションを表示したりして、剣を振る雰囲気を表現するゲームもあります。

剣を採用したゲームには、例えば「ドルアーガの塔」があります。このゲームでは、ボタンを押しっぱなしにすると、キャラクターが前方に剣を突き出します。剣が出ている間に、敵を突き刺すようにキャラクターを移動させると、敵にダメージを与えることができます。

「源平討魔伝」も剣を採用しています。このゲームにはいくつかのモードがありますが、キャラクターを大きく表示するモードでは、剣の画像を回転表示して、剣を振る様子を表現しています。レバーとの組み合わせによって、上段斬りや下段斬りなど、いくつかの方法で攻撃することができます。

「ゴールデンアックス」でも剣や斧を振るうことができます。姿は違いますが、使い方は剣も斧も同じです。このゲームでは、剣を振るう様子をアニメーションで表現しています。ジャンプ攻撃・ハイジャンプ攻撃・背後攻撃など、攻撃のバリエーションが多く、それぞれ違った剣の使い方が楽しめます。

「ラスタンサーガ」も剣を採用したゲームです。基本の武器は剣ですが、アイテムを拾うと斧なども使うことができます。普通に斬る以外にも、レバーを組み合わせることによって、上突きや下突きなどを出すことが可能です。

**Fig. 6-1** 剣を振る

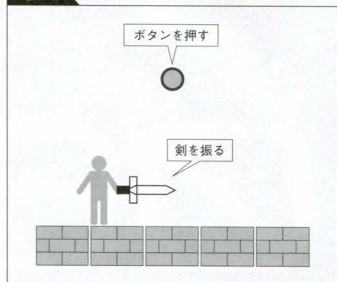
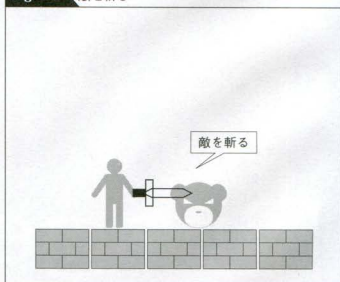



**Fig. 6-2** 敵を斬る

## アルゴリズム Algorithm

剣を実現するときのポイントは、剣を振る様子の表現です。例えば、剣の画像を回転表示させれば、比較的簡単に雰囲気を出すことができます。画像の表示位置を調整して、キャラクターが剣を手に持っている雰囲気を出すとよいでしょう。

もう1つのポイントは、敵を斬る処理です。これは剣と敵の当たり判定処理を行い、剣が敵に接触したら、敵にダメージを与えます。当たり判定処理は、剣の移動処理で行う方法と、敵の移動処理で行う方法がありますが、どちらを利用してもかまいません。

**Fig. 6-3** 剣の処理



## プログラム Program

List 6-1は剣のプログラムです。このサンプルでは、剣の画像を回転させて、剣を振る様子を表現しています。剣の移動処理や表示処理は、キャラクターの処理のなかで行う方法もありますが、このプログラムでは剣とキャラクターを別々のオブジェクトにしました。

また、剣と敵の当たり判定処理は、敵の移動処理で行うことにしました。敵が剣に接触したらダメージを受けます。この敵のオブジェクトは、本章のほかのサンプルでも使用しています。

**List 6-1** 剣（CWeaponEnemyクラス、CSwordManクラス、CSwordクラス）

```
// 敵の移動処理を行うMove関数
// この敵は剣だけではなく、ほかの武器のサンプルでも使用している
bool CWeaponEnemy::Move(const CInputState* is) {

	// 空中にいるときの加速度
	float accel=0.02f;

	// やられたときの回転スピード
	float vangle=0.05f;

	// 武器との当たり判定処理を行うための定数
	// X座標の差分の最大値
```

**List 6-1**

```
float max_dist=0.5f;

// やられたときの速度
float knockout_vx=0.4f;
float knockout_vy=-0.5f;

// 状態に応じて分岐する
switch (State) {

    // 通常状態
    case 0:

        // X座標の更新
        X+=VX;

        // 画面の左端から出たら、位置を初期化して再出現させる
        if (X<-1) Init();

        // 武器との当たり判定処理
        // 武器に接触したら、速度を設定して、やられ状態に移行する
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (
                mover->Type==1 &&
                abs(X-mover->X)<max_dist*mover->W &&
                abs(Y-mover->Y)<max_dist*mover->H
            ) {
                // 跳ね飛ばされた様子を表現するために、
                // 速度を設定する
                VX=knockout_vx;
                VY=knockout_vy;

                // やられ状態に移行する
                State=1;

                // 敵に当たると消える武器の場合には、
                // 武器を消去する
                if (RemoveWeapon) i.Remove();
                break;
            }
        }
        break;

    // やられ状態
    case 1:

        // X座標の更新
        X+=VX;

        // Y方向の速度を更新する
```

```
            VY+=accel;

            // Y座標の更新
            Y+=VY;

            // やられている様子を表すため、画像を回転させる
            Angle+=vangle;

            // 画面の右端から出たら、位置を初期化して再出現させる
            if (X>MAX_X) Init();
            break;
    }
    return true;
}

// キャラクターの移動処理を行うMove関数
bool CSwordMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 剣を出していないときの処理
    // Swordは剣を出しているかどうかを表す
    if (!Sword) {

        // レバーの入力に応じて左右に移動する
        // 攻撃の向きを決めるために、
        // キャラクターが移動した方向を保存しておく
        // VXとVYはキャラクターの速度を表す変数
        // DirXとDirYはキャラクターの移動方向を表す変数
        VX=0;
        if (is->Left) {
            DirX=-1;
            VX=-speed;
        }
        if (is->Right) {
            DirX=1;
            VX=speed;
        }

        // X座標を更新し、キャラクターが画面からはみ出さないように補正する
        X+=VX;
        if (X<0) X=0;
        if (X>MAX_X-1) X=MAX_X-1;

        // ボタンを押したら剣を出す
        // このサンプルでは剣のオブジェクトを作成する
        if (is->Button[0]) {
            new CSword(this);
            Sword=true;
```

**List 6-1**

```
		}
	}

	// X方向の速度に応じて、キャラクターを傾けて表示する
	Angle=VX/speed*0.1f;

	return true;
}

// 剣の移動処理を行うMove関数
bool CSword::Move(const CInputState* is) {

	// 剣を振るスピード
	float vangle=0.02f;

	// 剣の最大の角度
	float max_angle=0.5f;

	// 剣を振る様子を表現するために、画像を回転させる
	// キャラクターの向きに応じて回転方向を決める
	Angle+=vangle*Man->DirX;

	// 剣の角度に応じて、座標を計算する
	// キャラクターが剣の柄を持っているように、
	// 座標を調整する
	float rad=Angle*D3DX_PI*2;
	X=Man->X+Man->DirX*0.25f+sin(rad)*0.5f;
	Y=Man->Y-cos(rad)*0.5f;

	// 剣を最大の角度まで振り切ったら、
	// キャラクターのSwordメンバをfalseにして、
	// 剣のオブジェクトを消去する
	// 本書のサンプルではMove関数がfalseを返すと、
	// 呼び出し元の関数がオブジェクトを消去してくれる
	if (abs(Angle)>=max_angle) {
		Man->Sword=false;
		return false;
	}
	return true;
}
```

**SAMPLE**

「SWORD」は剣による攻撃アクションのサンプルです。レバー操作でキャラクターが左右に移動し、ボタンを押すと剣を振ります。剣が敵に当たると、敵はダメージを受けて吹き飛びます。

**SWORD** → p. 397

# ムチ

キャラクターの前方で伸び縮みする武器です。剣と同様に近接攻撃の一種ですが、伸び縮みするため、剣よりも遠い間合いから敵を攻撃することができます。

攻撃ボタンを押すと、キャラクターの前方にムチを出すことができます (Fig. 6-4)。ムチは前方へ伸びていき、完全に伸びると、その状態を一定時間維持します (Fig. 6-5)。

伸びきった状態から少し時間が経つと、ムチが縮み始めます (Fig. 6-6)。ムチが完全に縮んだら、攻撃は終了です。

ムチを敵に当てると、敵にダメージを与えることができます (Fig. 6-7)。ゲームによって違

**Fig. 6-4** ムチを出す

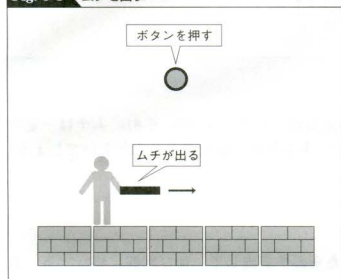
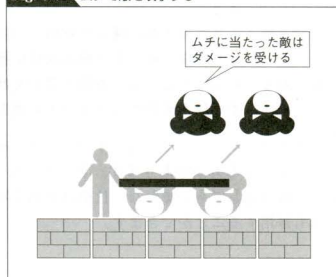



**Fig. 6-5** ムチが伸びきった状態

**Fig. 6-6** ムチが縮まった状態

**Fig. 6-7** ムチで敵を攻撃する

いますが、ムチが敵を貫通する場合には、1回ムチを振るうだけで複数の敵を倒すことが可能です。

ムチを採用したゲームには、例えば「悪魔城ドラキュラ」があります。このゲームでは、アイテムを取ると、ムチの長さが伸びたり、攻撃力がアップしたりします。

ムチではありませんが、「アルゴスの戦士」ではヨーヨーが使えます。動きとしてはムチとほぼ同じです。レバー入力を組み合わせると、ヨーヨーを半円形に振り回して、上からきた敵を攻撃することもできます。

「ポップフレイマー」には火炎放射器が登場します。キャラクターの前方に、ムチと同じ動きの炎を出すことができます。アイテムを取らないと、炎の長さがだんだん短くなります。

「最後の忍道」では鎖鎌が使えます。動きはムチと同じです。パワーアップすると円形に振り回せるようになります。

「海腹川背」には、「ロープを張る (→ p. 235)」でも紹介したように、伸び縮みするロープがあります。このロープはムチよりも多彩な用途に使えますが、間合いの離れた敵を攻撃するためにも使うことができます。

## アルゴリズム　Algorithm

ムチを実現するときには、まず敵との当たり判定処理が必要です (Fig. 6-8)。ムチは一定の長さがあるので、当たり判定もムチの形に合わせて、ある程度の幅を持たせるとよいでしょう。あるいは、ムチの先端だけに当たり判定を持たせる方法もあります。

もう1つのポイントは、ムチの状態を管理することです (Fig. 6-9)。ムチには次のような4つの状態があります。

- 通常状態：ムチを出す前の状態。ボタンを押したらムチを出して、伸び状態に移行する。あるいはムチのオブジェクトを作成する
- 伸び状態：時間とともにムチが伸びていく状態。ムチが最大長に達したら待機状態に移行する
- 待機状態：ムチが最大長に達した状態。一定時間が経過するまでは最大長の状態を維持する。少し時間が経ったら縮み状態に移行する
- 縮み状態：時間とともにムチが縮んでいく状態。ムチが完全に縮んだら通常状態に戻る。あるいはムチのオブジェクトを消去する

ムチのオブジェクトは、同じオブジェクトを使い回す方法と、ムチを出すたびに新しいオブジェクトを作る方法があります。タスクシステム (→ p. 7) のように、オブジェクトの作成と消去を繰り返してもかまわない仕掛けがある場合には、ムチを出すたびにオブジェクトを作る方法も採用することができます。

**Fig. 6-8** ムチの当たり判定

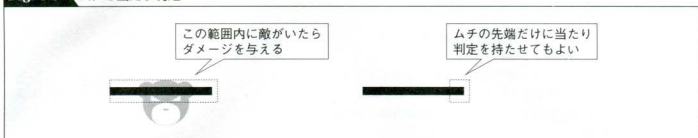
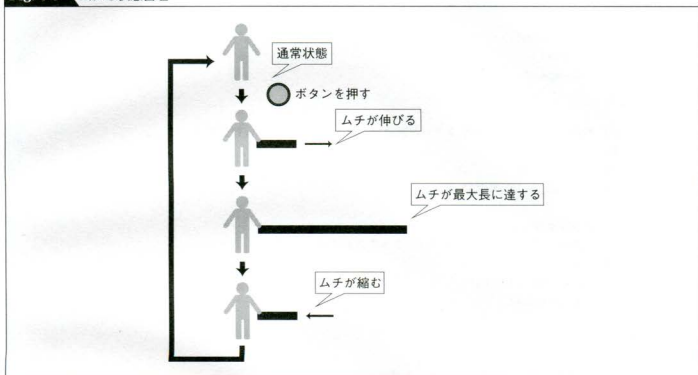



**Fig. 6-9** ムチの状態管理

## プログラム　Program

List 6-2はムチのプログラムです。このプログラムでは、ムチを出すたびにムチのオブジェクトを作り、ムチが完全に縮んだらオブジェクトを消去しています。

**List 6-2** ムチ(CWhipManクラス、CWhipクラス)

```
// キャラクターの移動処理を行うMove関数
bool CWhipMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // ムチを出していないときの処理
    // Whipはムチを出しているかどうかを表す
```

**List 6-2**

```
    if (!Whip) {

        // レバーの入力に応じて左右に移動する
        // 攻撃の向きを決めるために、
        // キャラクターが移動した方向を保存しておく
        // VXとVYはキャラクターの速度を表す変数
        // DirXとDirYはキャラクターの移動方向を表す変数
        VX=0;
        if (is->Left) {
            DirX=-1;
            VX=-speed;
        }
        if (is->Right) {
            DirX=1;
            VX=speed;
        }

        // X座標を更新し、キャラクターが画面からはみ出さないように補正する
        X+=VX;
        if (X<0) X=0;
        if (X>MAX_X-1) X=MAX_X-1;

        // ボタンを押したらムチを出す
        // このサンプルではムチのオブジェクトを作成する
        if (is->Button[0]) {
            new CWhip(this);
            Whip=true;
        }
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}

// ムチの移動処理を行うMove関数
bool CWhip::Move(const CInputState* is) {

    // ムチが伸び縮みするスピード
    float vlength=0.4f;

    // ムチの最大の長さ
    float max_length=4;

    // 状態に応じて分岐する
    switch (State) {

        // 伸び状態
        case 0:
```

```
            // 画像を横方向に引き伸ばして表示することによって、
            // ムチを伸ばす
            W+=vlength;

            // 最大の長さになったら、
            // 待機時間を設定して、待機状態に移行する
            if (W>=max_length) {
                Time=10;
                State=1;
            }
            break;

        // 待機状態
        case 1:

            // 待機時間を減らす
            Time--;

            // 時間が0になったら、縮み状態に移行する
            if (Time==0) State=2;
            break;

        // 縮み状態
        case 2:

            // 画像を横方向に縮めて表示することによって、
            // ムチを縮める
            W-=vlength;

            // 長さが0になったら、
            // キャラクターのWhipメンバをfalseにして、
            // ムチのオブジェクトを消去する
            if (W<=0) {
                Man->Whip=false;
                return false;
            }
            break;
    }

    // キャラクターがムチを手に持っているように、
    // 座標を調整する
    X=Man->X+Man->DirX*(W*0.5f+0.3f);
    Y=Man->Y;

    return true;
}
```

SAMPLE

「WHIP」はムチによる攻撃アクションのサンプルです。レバーでキャラクターを左右に動かし、ボタンでムチを出します。ムチが当たると、敵はダメージを受けて吹き飛びます。

**WHIP → p. 397**

# 跳ねるボール

壁で跳ね返るボールです。ボールを敵に当てて、ダメージを与えることができます。狭いところにボールを投げ入れると、壁に何度も跳ね返って面白い動きをします。

ボタンを押すと、キャラクターがボールを投げます (Fig. 6-10)。ボールは投げた方向に進みますが、壁に当たると跳ね返ります (Fig. 6-11)。ボールを敵に当てると、ダメージを与えて倒すことができます。

跳ねるボールを採用したゲームには、例えば「Mr.Do!」があります。このゲームでは、地中に掘った穴のなかにボールを投げることが可能です。ボールは壁に当たって跳ね返り、穴に沿って進んでいきます。敵にボールを当てると倒すことができます。

「ちゃっくんぽっぷ」では、爆弾を投げることができます。跳ね返るボールとは少し動きが違いますが、爆弾は床を跳ねながら転がっていきます。

Fig. 6-10 ボールを投げる

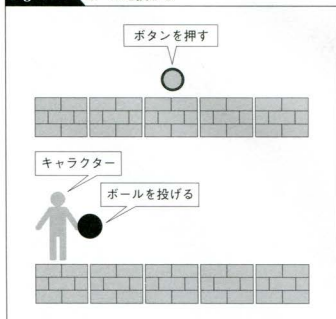
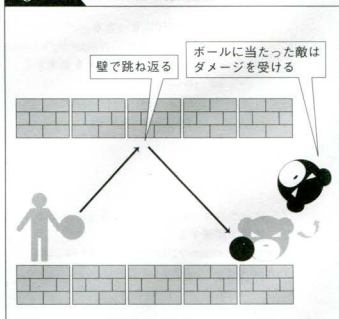
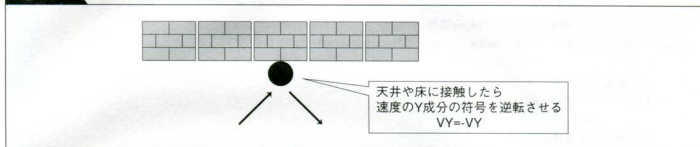



Fig. 6-11 ボールは壁で跳ね返る

## アルゴリズム Algorithm

跳ねるボールのポイントは、ボールを壁で跳ね返らせる処理です(Fig. 6-12)。壁との当たり判定処理を行い、例えば上下の壁(天井や床)に接触したら、速度のY成分の符号を反転させます。左右の壁で跳ね返らせるときには、左右の壁に接触したときに、速度のX成分の符号を反転させます。

**Fig. 6-12** ボールを跳ね返らせる処理

## プログラム Program

List 6-3は跳ねるボールのプログラムです。このサンプルでは、ボールは上下の壁で跳ね返るだけですが、上下左右の壁で跳ね返るようにもできます。その場合には、X座標を更新したあとに左右の壁との当たり判定処理を行い、続いて、Y座標を更新したあとに上下の壁との当たり判定処理を行えばよいでしょう。

**List 6-3** 跳ねるボール(CBouncingBallクラス、CBouncingBallManクラス)

```
// ボールの移動処理を行うMove関数
bool CBouncingBall::Move(const CInputState* is) {

    // 壁との当たり判定処理を行うための定数
    // Y座標の差分の最大値
    float max_dist=0.6f;

    // X座標とY座標の更新
    X+=VX;
    Y+=VY;

    // 壁との当たり判定処理
    // 壁に接触したら、速度のY成分の符号を反転させる
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==2 &&
            abs(X-mover->X)<max_dist &&
```

**List 6-3**

```
            abs(Y-mover->Y)<max_dist
        ) {
            VY=-VY;
            Y+=VY;
        }
    }

    // 画面の左右端から出たら、ボールを消去する
    return X>-1 && X<MAX_X;
}

// キャラクターの移動処理を行うMove関数
bool CBouncingBallMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // ボールを投げるスピード
    float throw_speed=0.2f;

    // レバーの入力に応じて左右に移動する
    // 攻撃の向きを決めるために、
    // キャラクターが移動した方向を保存しておく
    // VXとVYはキャラクターの速度を表す変数
    // DirXとDirYはキャラクターの移動方向を表す変数
    VX=0;
    if (is->Left) {
        DirX=-1;
        VX=-speed;
    }
    if (is->Right) {
        DirX=1;
        VX=speed;
    }

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // ボタンを押したら、
    // キャラクターの前方斜め上にボールを投げる
    if (!PrevButton && is->Button[0]) {
        new CBouncingBall(X, Y, DirX*throw_speed, -throw_speed);
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];
```

```
    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
  }
```

SAMPLE

「BOUNDING BALL」は跳ねるボールのサンプルです。レバーでキャラクターを左右に動かし、ボタンでボールを発射します。ボールは斜め上方向に向けて発射され、天井や床に跳ね返りながら進んでいきます。

**BOUNCING BALL** → p. 397

# 手榴弾

敵に向かって投げる爆弾です。投げた手榴弾は、着地すると爆発します。爆発に敵を巻き込めば、ダメージを与えることができます。

ボタンを押すと、キャラクターの前方に手榴弾（爆弾）を投げることができます（Fig. 6-13）。手榴弾は放物線を描いて飛んでいき、着地すると爆発します（Fig. 6-14）。この爆発に敵が接触するとダメージを受けます。

手榴弾を採用したゲームには、例えば「メタルスラッグ」があります。投げた手榴弾は、着地して少し転がったあとに爆発します。

「魔界村」のたいまつや「大魔界村」のナパームも、手榴弾に似た武器です。どちらも投げると放物線を描いて飛び、着地すると火柱を上げます。

**Fig. 6-13** 手榴弾を投げる

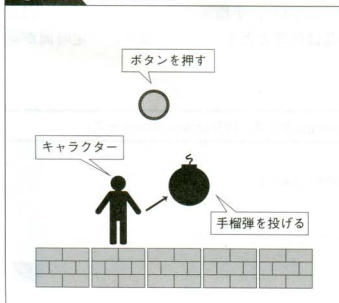



**Fig. 6-14** 手榴弾が爆発する

## アルゴリズム Algorithm

手榴弾のポイントは、爆発させる処理です。手榴弾が爆発する条件はゲームによって異なりますが、例えば次のようなものがあります。

・着地する
・敵に当たる
・一定距離を転がる

以上の条件が成立したら、手榴弾を消去して、かわりに爆発を生成します。そして、爆発に敵が接触したら敵にダメージを与えます（Fig. 6-15）。

放物線を描いて飛んでいく処理は、ジャンプの処理（→ p. 60）を応用して作ることができます。初速度や加速度などを調整すれば、手榴弾の速度や飛距離を変えることができます。

**Fig. 6-15** 手榴弾の処理

## プログラム Program

List 6-4は手榴弾のプログラムです。手榴弾が着地したら、手榴弾のオブジェクトを消去し、同じ座標に爆発のオブジェクトを生成します。爆発は時間とともに大きくなり、一定時間が経過すると消えます。

**List 6-4** 手榴弾（CGrenadeExplosionクラス、CGrenadeクラス、CGrenadeManクラス）

```
// 爆発の移動処理を行うMove関数
bool CGrenadeExplosion::Move(const CInputState* is) {

    // 幅と高さが大きくなるスピード
    float vsize=0.2f;

    // 幅と高さの最大値
```

```
    float max_size=4;

    // Y座標の調整
    Y-=vsize*0.5f;

    // 幅と高さの更新
    W+=vsize;
    H+=vsize;

    // 幅と高さが最大値になったら、爆発を消去する
    return W<max_size;
}

// 手榴弾の移動処理を行うMove関数
bool CGrenade::Move(const CInputState* is) {

    // 空中にいるときの加速度
    float accel=0.02f;

    // X座標の更新
    X+=VX;

    // Y方向の速度を更新する
    VY+=accel;

    // Y座標の更新
    Y+=VY;

    // 地面に落ちたら、爆発を生成し、手榴弾は消去する
    // 敵に当たった場合の処理は、
    // 敵の移動処理中に記述している
    if (Y>=MAX_Y-2) {
        new CGrenadeExplosion(X, Y);
        return false;
    }

    // 画面の左右端から出たら、手榴弾を消去する
    return X>-1 && X<MAX_X;
}

// キャラクターの移動処理を行うMove関数
bool CGrenadeMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 手榴弾を投げるスピード
    float throw_speed=0.2f;
```

↓

**List 6-4**

```
    // レバーの入力に応じて左右に移動する
    // 攻撃の向きを決めるために、
    // キャラクターが移動した方向を保存しておく
    // VXとVYはキャラクターの速度を表す変数
    // DirXとDirYはキャラクターの移動方向を表す変数
    VX=0;
    if (is->Left) {
        DirX=-1;
        VX=-speed;
    }
    if (is->Right) {
        DirX=1;
        VX=speed;
    }

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // ボタンを押したら、
    // キャラクターの前方斜め上に手榴弾を投げる
    if (!PrevButton && is->Button[0]) {
        new CGrenade(X, Y, DirX*throw_speed, -throw_speed*2);
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「GRENADE」は手榴弾のサンプルです。レバーでキャラクターを左右に動かし、ボタンで手榴弾（爆弾）を投げます。手榴弾は放物線状に飛んでいき、地面にぶつかったら爆発します。

**GRENADE** → p. 397

# 時限爆弾

一定時間が経過すると爆発する爆弾です。敵を爆発に巻き込むと、ダメージを与えることができます。

ボタンを押すと、キャラクターがいる位置に爆弾を設置します。爆弾は一定時間が経過すると爆発し、爆発に接触した敵はダメージを受けます(Fig. 6-17)。

時限爆弾を採用したゲームには、例えば「ボンバーマン」があります。このゲームでは、一定時間で爆発する時限爆弾を使って、敵を攻撃したり、壁を壊したりします。

「ワープ&ワープ」でも時限爆弾を使うことができます。このゲームではステージごとに攻撃方法が違いますが、一定時間で爆発する爆弾を使うステージがあります。

**Fig. 6-16** 爆弾の設置

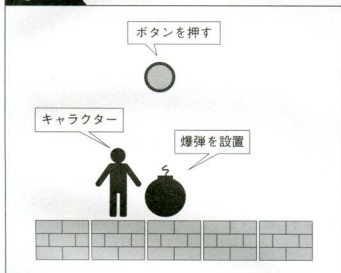



**Fig. 6-17** 爆弾の爆発

## アルゴリズム Algorithm

時限爆弾は、「手榴弾(→ p. 317)」と似た方法で実現することができます(Fig. 6-18)。手榴弾は飛んだり転がったりしますが、時限爆弾は設置した場所から動きません。また、着地したときではなく、一定時間が経過したときに爆発します。爆発したら、手榴弾の場合と同様に、爆弾を消去して、かわりに爆発を生成します。

## プログラム Program

List 6-5は時限爆弾のプログラムです。爆弾が動かないため、手榴弾よりもシンプルなプログラムになっています。

**List 6-5** 時限爆弾(CTimeBombクラス、CTimeBombManクラス)

```
// 爆弾の移動処理を行うMove関数
bool CTimeBomb::Move(const CInputState* is) {

    // 残り時間を減らす
    Time--;

    // 残り時間が0になったら、
    // 爆発を生成し、爆弾は消去する
    if (Time==0) {
        new CGrenadeExplosion(X, Y);
        return false;
    }

    return true;
}

// キャラクターの移動処理を行うMove関数
bool CTimeBombMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // ボタンを押したら、キャラクターがいる位置に爆弾を置く
    if (!PrevButton && is->Button[0]) {
        new CTimeBomb(X, Y);
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

「TIME BOMB」は時限爆弾のサンプルです。レバーでキャラクターを左右に動かし、ボタンで爆弾を設置します。爆弾は一定時間が経過すると爆発し、爆発に触れた敵はダメージを受けます。

**TIME BOMB** → p. 398

## 爆煙

爆弾を爆発させると、煙が広がるアクションです。煙に敵を巻き込めば、ダメージを与えることができます。また、煙は壁と壁の間をぬうように広がっていくため、見ていて面白い動きになります。

最初にボタンを押して、キャラクターがいる位置に爆弾を置きます (Fig. 6-18)。一定時間が経過すると爆弾は爆発して、爆煙が発生します (Fig. 6-19)。

時間とともに爆煙は広がっていき (Fig. 6-20)、一定時間が経過すると爆煙は消えます。爆煙に敵を巻き込むと、敵にダメージを与えることができます。

爆煙で面白いのは、壁などの障害物を避けて広がっていく点です (Fig. 6-21)。迷路のように複雑な地形で爆煙を使うと、爆煙が通路に沿って広がっていく様子が見られます。

爆煙を採用したゲームには、例えば「ちゃっくんぽっぷ」があります。このゲームでは、爆弾が爆発すると爆煙が発生します。爆煙は時間とともに広がり、一定時間で消えます。爆煙は壁やほかの爆煙をよけて広がるため、爆弾を狭いところで爆発させたり、2個以上の爆弾をいっしょに爆発させたりすると、爆煙が面白い動きをします。この動きを利用して、爆煙を遠くの敵まで届かせるテクニックもあります。

**Fig. 6-18** 爆弾の設置

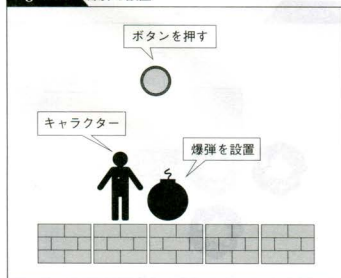



**Fig. 6-19** 爆煙の発生

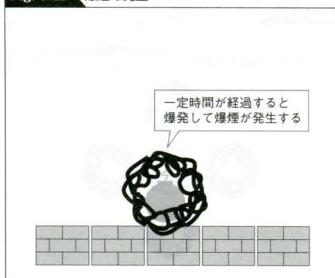



Fig. 6-20 爆煙が広がる

Fig. 6-21 爆煙は壁を避けて広がる

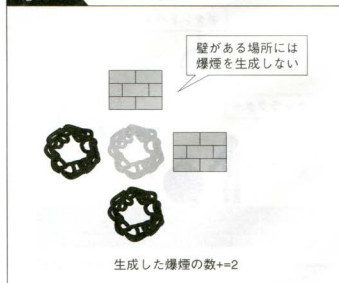
## アルゴリズム Algorithm

爆煙のポイントは、爆煙の生成処理です(Fig. 6-22)。1個の爆煙の四方に新しい爆煙を生成します。そして、新しい爆煙がまた別の爆煙を生成し、その爆煙がまた爆煙を生成する…という処理を繰り返すと、広がる爆煙を実現することができます。

生成した爆煙の数は記録しておく必要があります。生成可能な爆煙の数に上限を設けておかないと、爆煙が画面全体を埋め尽くすまで広がってしまうからです。四方に爆煙を生成した場合には、爆煙の数はプラス4個になります。

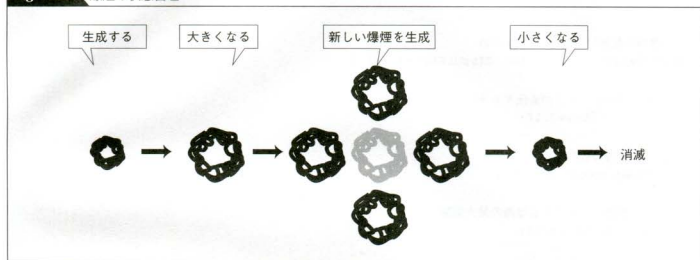
爆煙を生成するときには、四方を調べて、壁やほかの爆煙がある場所には生成しないようにします(Fig. 6-23)。例えば、上と右に壁がある場合は、左と下にだけ爆煙を生成します。そう

Fig. 6-22 爆煙の生成処理

Fig. 6-23 壁を避ける処理

すると、爆煙の生成数はプラス2個になり、四方に爆煙を生成する場合（プラス4個）よりも爆煙の生成を多く繰り返すことができます。

これで、爆煙は壁やほかの爆煙をよけて広がるようになります。そのため、複数の爆弾を同時に（厳密にはいくらかのタイムラグはありますが）爆発させると、爆煙をよけながら爆煙が広がっていくので、通常よりも広い範囲を攻撃することができます。

1個の爆煙に関する処理は、Fig. 6-24のような手順で行います。生成した爆煙は、時間とともに大きくなります。爆煙が最大のサイズになったら、四方に新しい爆煙を生成します。中央の爆煙は時間とともに緩やかに小さくなり、最後は消滅します。

**Fig. 6-24** 爆煙の状態管理

## プログラム Program

List 6-6は爆煙のプログラムです。このサンプルでは、「時限爆弾（→ p. 321）」から爆煙が発生します。時限爆弾のかわりに、「手榴弾（→ p. 317）」などから爆煙を発生させることもできます。

**List 6-6** 爆煙（CBlastクラス、CBlastBombクラス、CBlastManクラス）

```
// 爆煙と壁の当たり判定処理を行うHit関数
// 引数で指定された座標(x, y)が、壁に接触するかどうかを調べる
bool CBlast::Hit(float x, float y) {

    // 壁およびほかの爆煙との当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=0.5f;

    // 壁およびほかの爆煙との当たり判定処理
    // 壁またはほかの爆煙に接触したらtrueを返す
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
```

**List 6-6**

```
            CMover* mover=(CMover*)i.Next();
            if (
                mover->Type!=0 &&
                abs(x-mover->X)<max_dist &&
                abs(y-mover->Y)<max_dist
            ) {
                return true;
            }
        }

        // 接触しなかったらfalseを返す
        return false;
    }

    // 爆煙の移動処理を行うMove関数
    bool CBlast::Move(const CInputState* is) {

        // 爆煙のサイズが変化するスピード
        float vsize=0.1f;

        // 爆煙の最大サイズ
        float max_size=1.0f;

        // 画面上に存在する爆煙の最大個数
        int max_blast=20;

        // 状態に応じて分岐する
        switch (State) {

            // 拡大状態
            case 0:

                // サイズを大きくする
                W+=vsize;
                H+=vsize;

                // 爆煙が最大サイズになったときの処理
                if (W>=max_size) {

                    // 上下左右に新しい爆煙を生成する
                    // Hit関数を使って壁やほかの爆煙があるかどうかを調べ、
                    // 壁やほかの爆煙がある位置には生成しない
                    // また、爆煙の生成数が最大個数に達していたら、
                    // それ以上爆煙を生成しない
                    // 爆煙の生成数は、
                    // 爆弾オブジェクトのBlastCountメンバに記録する
                    for (int i=-1; i<=1; i+=2) {
                        if (Bomb->BlastCount<max_blast && !Hit(X+i, Y)) {
                            new CBlast(X+i, Y, Bomb);
                        }
```

```
                if (Bomb->BlastCount<max_blast && !Hit(X, Y+i)) {
                    new CBlast(X, Y+i, Bomb);
                }
            }

            // 縮小状態に移行する
            State=1;
        }
        break;

    // 縮小状態
    case 1:

        // サイズを小さくする
        // 拡大時よりも緩やかに縮小する
        W-=vsize*0.1f;
        H-=vsize*0.1f;

        // サイズが0になったら、爆煙を消去する
        // 爆弾のBlastCountメンバを減少させる
        if (W<=0) {
            Bomb->BlastCount--;
            return false;
        }
        break;
    }
    return true;
}

// 爆弾の移動処理を行うMove関数
bool CBlastBomb::Move(const CInputState* is) {

    // 残り時間を減少させる
    Time--;

    // 残り時間が0になったら爆発させる
    if (Time==0) {

        // 爆弾の位置に、1個の爆煙を生成する
        new CBlast(X, Y, this);

        // 爆弾の幅と高さを0にして、画面から消去する
        W=H=0;
    }

    // 残り時間が0以上、またはBlastCountが1以上の間は、
    // 爆弾オブジェクトを残存させる
    // BlastCountは画面上にある爆煙の数を表す
    // 爆煙は爆弾のBlastCountメンバを参照するため、
    // すべての爆煙オブジェクトを消去したあとでないと、
```

**List 6-6**

```
    // 爆弾オブジェクトを消去することができない
    // そこで、BlastCountメンバを使って、
    // 爆煙がいくつ残っているのかを数える
    return Time>=0 || BlastCount>0;
}

// キャラクターの移動処理を行うMove関数
bool CBlastMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // ボタンを押したら、キャラクターがいる位置に爆弾を置く
    if (!PrevButton && is->Button[0]) {
        new CBlastBomb(X, Y);
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「BLAST」は爆煙のサンプルです。レバーでキャラクターを左右に動かし、ボタンで爆弾を設置します。爆弾は一定時間が経過すると爆発し、爆煙を発します。爆煙に触れた敵はダメージを受けます。

**BLAST** → p. 398

# マシンガン

ボタンを押している間、自動的に弾丸を連射する武器です。使いやすく強力な武器なので、威力を弱めにしたり、射程を少し短めにするなどして、バランスをとっているゲームもあります。

マシンガンを撃つには、ボタンを押します(Fig. 6-25)。ボタンを押したままにしている間、弾丸が連射されます。

弾丸に当たった敵はダメージを受けます(Fig. 6-26)。多くのゲームでは、敵に当たった弾丸は貫通せずに消えます。

マシンガンを採用したゲームには、例えば「魂斗羅」があります。このゲームでは、レバー入力を組み合わせて、キャラクターの上や斜め上にもマシンガンを撃つことができます。ほかにも、キャラクターではなく敵が撃ってくる場合も含めて、多くのゲームでマシンガンが採用されています。

**Fig. 6-25** マシンガン

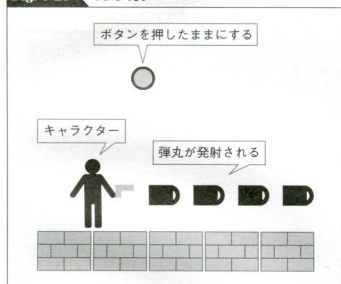



**Fig. 6-26** 敵に弾丸を当てる

## アルゴリズム Algorithm

マシンガンのポイントは、弾丸を生成する処理です(Fig. 6-27)。ボタンが押されている間、一定時間ごとに弾丸を生成します。弾丸を生成してから、次の弾丸を生成するまでに一定時間をおくことによって、弾丸と弾丸の間隔がつまりすぎてしまうのを防ぎます。弾丸が速いときには時間を短めに、遅いときには時間を長めにとると、弾丸の間隔が適度な広さになるでしょう。

もう1つのポイントは、弾丸が飛んだ距離を記録しておき、一定距離を飛んだ弾丸は消すことです。これで弾丸の射程を制限することができます。マシンガンは強力な武器なので、少し射程を短めにしておいた方が、敵に接近して撃つ必要が生じて、ゲームが面白くなります。

**Fig. 6-27** マシンガンの処理

## プログラム Program

List 6-7はマシンガンのプログラムです。このプログラムでは、5フレーム（約1/12秒）ごとに弾丸を生成しています。弾丸のスピードに応じて、画面上で弾丸がきれいに並ぶように、発射の間隔を調整するとよいでしょう。

**List 6-7** マシンガン（CMachineGunBulletクラス、CMachineGunManクラス）

```
// 弾丸の移動処理を行うMove関数
bool CMachineGunBullet::Move(const CInputState* is) {

    // 最大の移動距離
    float max_dist=4;

    // X座標の更新
    X+=VX;

    // 移動距離の更新
    Dist+=abs(VX);

    // 移動距離が最大値に達したら、弾を消去する
    return Dist<max_dist;
}

// キャラクターの移動処理を行うMove関数
bool CMachineGunMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 弾のスピード
    float bullet_speed=0.2f;

    // 弾を撃つ間隔（フレーム数）
    int wait_time=5;
```

```
        // レバーの入力に応じて左右に移動する
        // 攻撃の向きを決めるために、
        // キャラクターが移動した方向を保存しておく
        // VXとVYはキャラクターの速度を表す変数
        // DirXとDirYはキャラクターの移動方向を表す変数
        VX=0;
        if (is->Left) {
            DirX=-1;
            VX=-speed;
        }
        if (is->Right) {
            DirX=1;
            VX=speed;
        }

        // X座標を更新し、キャラクターが画面からはみ出さないように補正する
        X+=VX;
        if (X<0) X=0;
        if (X>MAX_X-1) X=MAX_X-1;

        // 次の弾を撃つまでの残り時間が0より大きいときには、時間を減少させる
        if (Time>0) Time--;

        // 残り時間が0のときにボタンを押していたら、
        // マシンガンを撃つ
        // 弾を生成し、残り時間を設定する
        if (Time==0 && is->Button[0]) {
            new CMachineGunBullet(X, Y, DirX*bullet_speed);
            Time=wait_time;
        }

        // X方向の速度に応じて、キャラクターを傾けて表示する
        Angle=VX/speed*0.1f;

        return true;
    }
```

SAMPLE

「MACHINE GUN」はマシンガンのサンプルです。レバーでキャラクターを左右に動かし、ボタンでマシンガンを発射します。ボタンを押し続けている間、マシンガンを連射します。マシンガンの弾は、一定距離を飛ぶと消えます。

**MACHINE GUN** → p. 398

## 誘導ミサイル

目標を自動的に追尾するミサイルです。ミサイルが滑らかな曲線を描いて敵を追尾する様子は、見ていて楽しい動きです。シューティングゲームに多く採用されている武器ですが、アクションゲームにもよく登場します。

最初に、ボタンを押してミサイルを発射します(Fig. 6-28)。発射されたミサイルは緩やかに方向を変えながら、敵に向かって誘導されます(Fig. 6-29)。どの敵を狙うかはゲームによって違いますが、ここでは一番近い敵に誘導されることにしました。着地したり、敵に当たったりすると、ミサイルは爆発します。爆発に敵を巻き込むと、ダメージを与えることができます。

誘導ミサイルを採用したゲームには、例えば「チェルノブ」があります。誘導ミサイルはいかにも強力そうに思える武器ですが、このゲームではミサイルの連射が利かないため、ほかの武器よりもかえって弱い武器という位置づけになっています。ゲーム全体のバランスをとるため、そして使って楽しい誘導ミサイルにするためには、ミサイルの攻撃力や連射数などを念入りに調整する必要があります。

**Fig. 6-28** ミサイルの発射

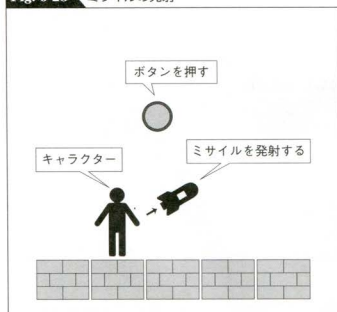
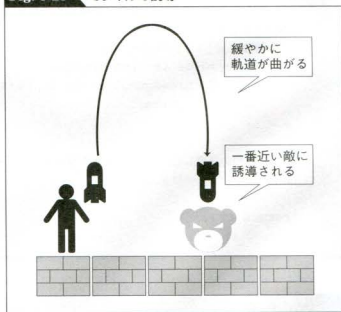



**Fig. 6-29** ミサイルの誘導

### アルゴリズム Algorithm

誘導ミサイルのポイントは、ミサイルを目標に誘導する処理です。単に目標の方向に進ませるだけでは、ミサイルは緩やかな曲線を描きません。また、必ず目標に命中するミサイルになってしまっては面白みに欠けます。そこで、目標の方向とミサイルの現在の進行方向の両方を考慮して、目標の方向に近づくようにミサイルを少しずつ回転させる処理が必要になります。

例えば、目標の方向がミサイルの進行方向に対して右側にあるときには、ミサイルを右回りに回転させます（Fig. 6-30）。ミサイルの座標を（X, Y）、進行方向をRad、目標の座標を（tx, ty）とすると、進行方向と敵の方向がなす角度diffは、

```
diff=atan2(ty-Y, tx-X)-Rad
```

となります（Fig. 6-31）。この角度diffが0度から180度、ラジアンでは0からπの範囲にあるときには、ミサイルを右回りに回転させます。ミサイルの回転スピードをvradとすると、ミサイルの新しい進行方向は、

**Fig. 6-30** ミサイルの進行方向

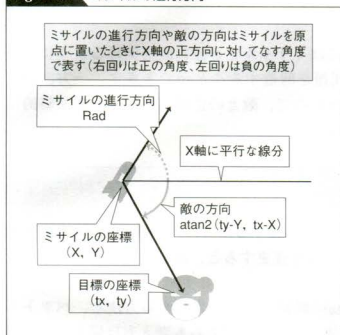



**Fig. 6-31** ミサイルを右回りに回転させる

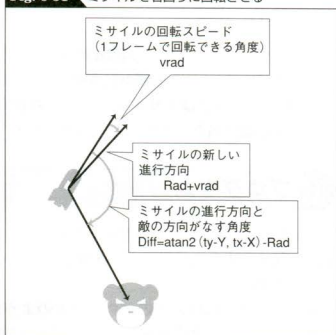



**Fig. 6-32** ミサイルを左回りに回転させる

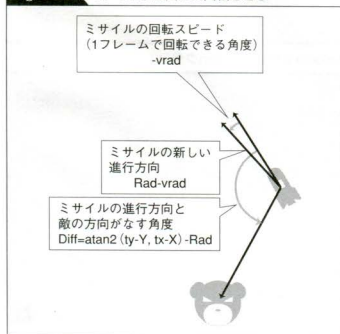
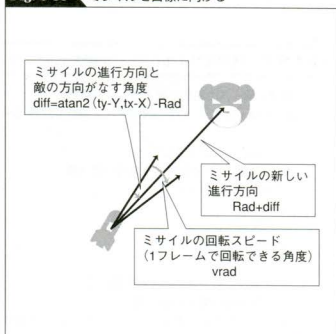



**Fig. 6-33** ミサイルを目標に向ける

```
Rad+vrad
```

となります。

角度diffが-180度から0度、ラジアンでは-πから0の範囲にあるときには、ミサイルを左回りに回転させます（Fig. 6-32）。ミサイルの新しい進行方向は、

```
Rad-vrad
```

となります。

また、角度diffの絶対値が、ミサイルの回転スピードvradの絶対値よりも小さいときには、ミサイルを目標の方向に直接向けます（Fig. 6-33）。このときの新しい進行方向は、

```
Rad+diff
```

となります。このように、角度diffが小さいときにはミサイルを目標に直接向けるようにすると、ミサイルが左右に小刻みに振動してしまう問題を回避することができます。一方、角度diffが大きいときにはミサイルを目標に直接向けないので、敵との位置関係によって最終的にミサイルが敵に当たったり当たらなかったりします。

## プログラム　Program

List 6-8は誘導ミサイルのプログラムです。処理を簡単にするために、角度diffは-πからπの範囲に収まるように補正しています。また、vradの値を変更すると、誘導の強さを変えることができます。

誘導ミサイルには、このプログラムのようにatan2関数を使って実現する方法と、ベクトルを使って実現する方法があります。処理の詳細は違いますが、どちらも考え方は同じです。ベクトルを使った方法については、拙著『シューティングゲームアルゴリズムマニアックス』で紹介していますので、興味がある方はそちらをご参照ください（→ p. 8）。

**List 6-8**　誘導ミサイル（CHomingMissileクラス、CHomingMissileManクラス）

```
// 誘導ミサイルの移動処理を行うMove関数
bool CHomingMissile::Move(const CInputState* is) {

    // 方向を変えるスピード
    // 1フレームで変化する角度の最大値
    float vrad=0.08f;

    // 移動スピード
    float speed=0.2f;

    // すべての敵までの距離を調べて、
    // 最も近い敵を目標に選ぶ
    // 目標に選んだ敵へのポインタはtargetに格納する
```

```
float target_dist=MAX_X+MAX_Y;
CMover* target=NULL;
for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
    CMover* mover=(CMover*)i.Next();
    if (mover->Type==2) {
        float
            dx=abs(X-mover->X),
            dy=abs(Y-mover->Y),
            dist=sqrt(dx*dx+dy*dy);
        if (dist<target_dist) {
            target_dist=dist;
            target=mover;
        }
    }
}

// 目標に対してミサイルを誘導する処理
if (target) {

    // 円周率の2倍の値角度の計算に用いる
    float pi2=D3DX_PI*2;

    // 目標の方向と現在のミサイルの方向がなす角度diffを求める
    float diff=atan2f(target->Y-Y, target->X-X)-Rad;

    // diffを-πラジアンからπまでの範囲に補正する
    while (diff<-D3DX_PI) diff+=pi2;
    while (diff>=D3DX_PI) diff-=pi2;

    // diffが1フレームで変化できる範囲ならば、
    // ミサイルを目標の方向に直接向ける
    if (abs(diff)<vrad) {
        Rad+=diff;
    } else

    // diffが1フレームで変化できる範囲を超えていたら、
    // ミサイルの方向が目標の方向に近づくように、
    // 角度に定数を加える
    {
        Rad+=(diff<0?-vrad:vrad);
    }
}

// 座標の更新
// 向いている方向にミサイルを進ませる
X+=cos(Rad)*speed;
Y+=sin(Rad)*speed;

// ミサイルが地面に達したら爆発させる
// 爆発を生成し、ミサイルは消去する
```

List 6-8

```
    if (Y>=MAX_Y-2) {
        new CGrenadeExplosion(X, Y);
        return false;
    }

    // ミサイルの画像を進行方向に合わせて回転させる
    Angle=Rad/D3DX_PI/2+0.25f;

    // 画面の左右端から出たら、ミサイルを消去する
    return X>-1 && X<MAX_X;
}

// キャラクターの移動処理を行うMove関数
bool CHomingMissileMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // ボタンを押したらミサイルを発射する
    if (!PrevButton && is->Button[0]) {
        new CHomingMissile(X, Y, -D3DX_PI/2);
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「HOMING MISSILE」は誘導ミサイルのサンプルです。レバーでキャラクターを左右に動かし、ボタンでミサイルを発射します。ミサイルは一番近い敵に向かって飛んでいきます。

**HOMING MISSILE** → p. 398

# ブーメラン

投げると遠くに飛んでいき、一定時間飛行すると、反転して手元に戻ってくる武器です。飛行中のブーメランに当たった敵を、まとめて倒すことができます。

ボタンを押すと、ブーメランを投げることができます(Fig. 6-34)。投げたブーメランはしばらく飛んだあとに、緩やかにターンします。そして、キャラクターに近づいてきて、手元に戻ります(Fig. 6-35)。飛んでいるブーメランが敵に当たると、ダメージを与えることができます。

ブーメランを採用したゲームには、例えば「ゼルダの伝説」があります。このゲームでは、まとめてロックした複数の敵にブーメランを当てたり、ブーメランでアイテムを回収したりといったアクションが楽しめます。また、「チェルノブ」にもブーメランが登場します。

**Fig. 6-34** ブーメラン

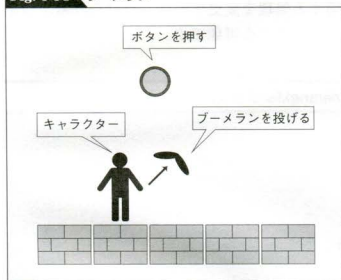
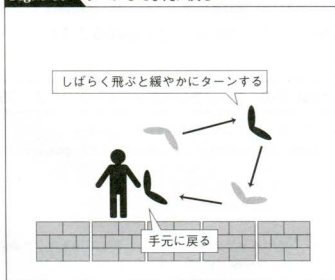
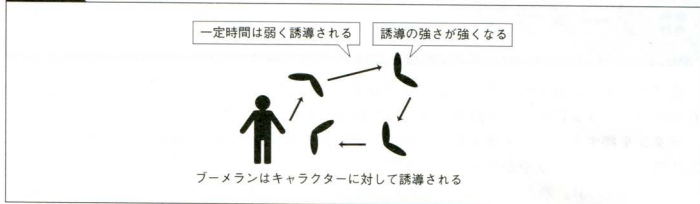



**Fig. 6-35** ターンして手元に戻る

## アルゴリズム Algorithm

ブーメランのポイントは、キャラクターの手元に返ってくる動きです。これは、キャラクターに対してブーメランを誘導することによって実現します(Fig. 6-36)。誘導の処理は、「誘導ミサイル(→ p. 332)」の場合とまったく同じです。ただし、ブーメランの場合には、時間とともに誘導の強さを変化させます。ブーメランの角度が変化するスピード、つまり1フレームあたりに回転できる角度を変更すれば、誘導の強さを変えることができます。

最初はブーメランをキャラクターに対して弱く誘導します。すると、ブーメランは緩やかな曲線を描きながら、キャラクターから離れていきます。一定時間が経過したら、ブーメランの誘導を強くします。すると、ブーメランはキャラクターに近づく方向へ反転して、キャラクターの手元に戻ってきます。

**Fig. 6-36** ブーメランの処理

## プログラム Program

List 6-9はブーメランのプログラムです。このサンプルでは、一度に投げられるブーメランを1本だけに制限しました。Boomerangメンバに関する処理を変更すれば、投げられる本数を増やしたり、無制限に投げられるようにしたりといったことも可能です。

**List 6-9** ブーメラン(CBoomerangクラス、CBoomerangManクラス)

```
// ブーメランの移動処理を行うMove関数
bool CBoomerang::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // キャラクターとの当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=0.6f;

    // 残り時間が0より大きければ、時間を減らす
    if (Time>0) Time--;

    // 残り時間が0より大きいときには誘導を弱く、
    // 0のときには誘導を強くする
    float vrad=(Time==0?0.1f:0.02f);

    // 誘導ミサイルと同じ方法で、
    // ブーメランをキャラクターに向かって誘導する
    float pi2=D3DX_PI*2;
    float diff=atan2f(Man->Y-Y, Man->X-X)-Rad;
    while (diff<-D3DX_PI) diff+=pi2;
    while (diff>=D3DX_PI) diff-=pi2;
    if (abs(diff)<vrad) {
        Rad+=diff;
    } else {
```

```
		Rad+=(diff<0?-vrad:vrad);
	}

	// 座標の更新
	X+=cos(Rad)*speed;
	Y+=sin(Rad)*speed;

	// 進行方向に合わせてブーメランの画像を回転させる
	Angle=Rad/D3DX_PI/2+0.25f;

	// 返ってきたブーメランがキャラクターに接触したら、
	// キャラクターのBoomerangメンバをfalseにして、
	// ブーメランを消去する
	// Boomerangメンバは、
	// ブーメランが手元にあるかどうかを判定するために使う
	// ブーメランを投げる前はfalseに、投げているときにはtrueにする
	if (
		Time==0 &&
		abs(Man->X-X)<max_dist &&
		abs(Man->Y-Y)<max_dist
	) {
		Man->Boomerang=false;
		return false;
	}
	return true;
}

// キャラクターの移動処理を行うMove関数
bool CBoomerangMan::Move(const CInputState* is) {

	// 移動スピード
	float speed=0.2f;

	// レバーの入力に応じて左右に移動する
	// 攻撃の向きを決めるために、
	// キャラクターが移動した方向を保存しておく
	// VXとVYはキャラクターの速度を表す変数
	// DirXとDirYはキャラクターの移動方向を表す変数
	VX=0;
	if (is->Left) {
		DirX=-1;
		VX=-speed;
	}
	if (is->Right) {
		DirX=1;
		VX=speed;
	}

	// X座標を更新し、キャラクターが画面からはみ出さないように補正する
	X+=VX;
```

**List 6-9**

```
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // ブーメランが手元にあるときにボタンを押したら、
    // ブーメランを投げる
    // Boomerangメンバをtrueにして、
    // ブーメランを生成する
    if (!Boomerang && is->Button[0]) {
        Boomerang=true;
        new CBoomerang(X, Y, (DirX<0?D3DX_PI*6/5:-D3DX_PI/5), this);
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「BOOMERANG」はブーメランのサンプルです。レバーでキャラクターを左右に動かし、ボタンでブーメランを発射します。ブーメランは斜め上方向に向かって発射され、曲線を描きながらキャラクターの手元に戻ってきます。

**BOOMERANG** → p. 337

# リモコン武器

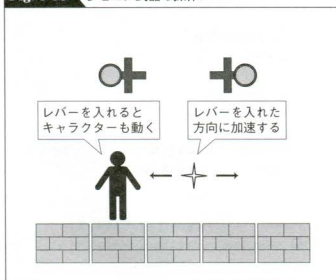
レバー操作で動かすことができる武器です。キャラクターと武器を同時に動かすことになるため、自在に動かすにはテクニックが必要です。

ボタンを押すとリモコン武器が出現します(Fig. 6-37)。ここでは手裏剣の形にしました。リモコン武器は、レバー操作などで動かすことができます。ここでは、レバーを入れた方向にリモコン武器が加速することにしました(Fig. 6-38)。レバーを入力するとキャラクターも同時に動きますが、キャラクターとリモコン武器とは微妙に違った動きをします。

リモコン武器を敵に当てると、ダメージを与えることができます。ある程度は思ったように動くけれども意のままに操るのは難しい、といったバランスにしておくと、面白い操作感覚になるでしょう。

リモコン武器を採用したゲームには、例えば「ゲゲゲの鬼太郎 妖怪大魔境」があります。このゲームでは下駄をリモコン操作することができます。リモコン操作はレバーで行うため、武器と同時にキャラクターも動きます。そのため、武器の操作に気を取られすぎると、キャラクターが落とし穴に落ちたり、敵に接触したりする危険があります。

**Fig. 6-37** リモコン武器を出す

**Fig. 6-38** リモコン武器の操作

## アルゴリズム Algorithm

リモコン武器を実現するには、レバーの入力に応じて武器を動かします。これはレバーでキャラクターを動かす処理と同じです。ただし、武器とキャラクターをまったく同じ方法で動かすと、武器とキャラクターが重なったまま動いてしまいます。微妙に武器とキャラクターの動きを変えることが、面白いリモコン武器を作るためのポイントです。

## プログラム Program

List 6-10はリモコン武器のプログラムです。リモコン武器の移動処理では、キャラクターの移動処理と同様に、レバー入力に応じて武器を動かします。武器の移動には加速度を使っているので、キャラクターとは微妙に違った動きをします。

**List 6-10** リモコン武器(CRemoteControlWeaponクラス、CRemoteControlManクラス)

```
// リモコン武器の移動処理を行うMove関数
bool CRemoteControlWeapon::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 加速度
    float accel=0.04f;

    // レバーを入力に応じて、
    // 速度に対して左右に加速度を加える
    if (is->Left && VX>-speed) VX-=accel;
```

**List 6-10**

```
    if (is->Right && VX<speed) VX+=accel;

    // X座標の更新
    X+=VX;

    // 画像の回転
    Angle+=0.01f;

    // 画面の左右端から出たら、武器を消去する
    return X>-1 && X<MAX_X;
}

// キャラクターの移動処理を行うMove関数
bool CRemoteControlMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.1f;

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // ボタンを押したらリモコン武器を出す
    if (!PrevButton && is->Button[0]) {
        new CRemoteControlWeapon(X, Y);
    }

    // ボタンを押した瞬間を判定するために、
    // 現在のボタンの状態を保存しておく
    PrevButton=is->Button[0];

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「REMOTE CONTROL」はリモコン武器のサンプルです。レバーでキャラクターを左右に動かし、ボタンでリモコン武器（手裏剣）を発射します。リモコン武器は左右のレバーで操作することができます。

**REMOTE CONTROL** → p. 398

# 盾

キャラクターがかまえることができる盾です。敵の攻撃を防いだり、弾丸を跳ね返したりといった効果があります。

ボタンを押すと、キャラクターが盾をかまえます (Fig. 6-39)。ボタンを押したままにしている間、キャラクターは盾をかまえ続けます。盾をかまえていると、敵の攻撃を防いだり、弾丸を跳ね返したりすることができます (Fig. 6-40)。

盾を採用したゲームには、例えば「戦いの挽歌」があります。このゲームでは、剣・斧・矢といった敵の攻撃を、盾を使って防ぐことができます。敵の攻撃には上段攻撃と下段攻撃があるため、「しゃがむ (→ p. 291)」アクションを併用して、盾を適切な高さにかまえる必要があります。

Fig. 6-39 盾をかまえる

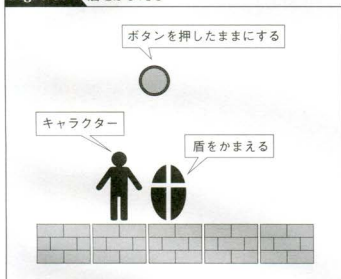



Fig. 6-40 攻撃を跳ね返す

## アルゴリズム Algorithm

盾を実現する方法は簡単です。ボタンの入力を調べて、ボタンが押されている間に盾を出すだけです。盾のオブジェクトを生成し、それに敵の攻撃が接触したら、攻撃を無効にしたり、跳ね返したりといった処理を行います。

## プログラム Program

List 6-11は盾のプログラムです。このプログラムでは、盾をキャラクターとは別のオブジェクトにしました。盾を出すときには盾のオブジェクトを生成し、盾を引っ込めたらオブジェクトを消去します。

このようにオブジェクトの生成と消去を行うのではなく、例えば盾を出しているかどうかのフラグを用意して、フラグに応じて盾の当たり判定処理や描画処理を行う、という方法でも実現できます。

**List 6-11** 盾(CShieldManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 盾のX座標
    // キャラクターのX座標に対する相対値
    float shield_x=0.6f;

    // レバーの入力に応じて左右に移動する
    // 盾を出す向きを決めるために、
    // キャラクターが移動した方向を保存しておく
    // VXとVYはキャラクターの速度を表す変数
    // DirXとDirYはキャラクターの移動方向を表す変数
    VX=0;
    if (is->Left) {
        DirX=-1;
        VX=-speed;
    }
    if (is->Right) {
        DirX=1;
        VX=speed;
    }

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // まだ盾を出していなければ、盾を出す
    // Shieldは盾へのポインタを保持する
    // 盾を出しているときには、ShieldがNULL以外の値になる
    if (!Shield && is->Button[0]) Shield=new CShield();

    // 盾を出しているときの処理
    if (Shield) {

        // キャラクターが盾を構えているように、
        // 盾の座標を設定する
        Shield->X=X+DirX*shield_x;
        Shield->Y=Y;
```

```
        // ボタンを放したら、盾を消去する
        if (!is->Button[0]) {
            delete Shield;
            Shield=NULL;
        }
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「SHIELD」は盾をかまえるアクションのサンプルです。レバーでキャラクターを左右に動かし、ボタンで盾をかまえます。ボタンを押している間は盾をかまえ続けます。盾で飛んでくる弾を防ぐことができます。

**SHIELD** → p. 398

# 煙幕

キャラクターが移動しながら煙幕を張る攻撃です。煙幕に敵を巻き込めば、ダメージを与えたり、行動不能にさせたりできます。

ボタンを押すとキャラクターが煙幕を張ります (Fig. 6-41)。レバー操作と組み合わせて、煙幕を張りながら移動することもできます。

**Fig. 6-41** 煙幕を張る

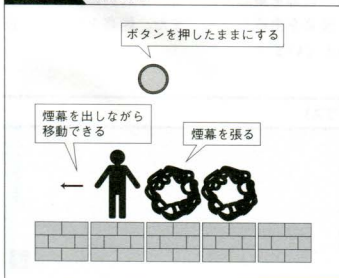
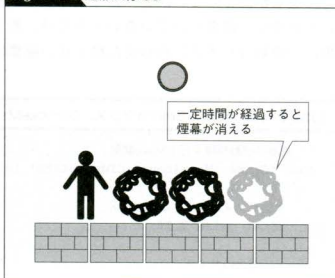
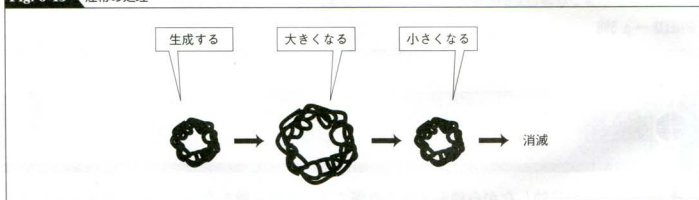



**Fig. 6-42** 煙幕が消える

煙幕は一定時間が経過すると消えます(Fig. 6-42)。連続して煙幕を張り続けたときには、最初に張った煙幕からだんだん消えていきます。煙幕に当たった敵はダメージを受けます。ゲームによっては、敵が行動不能になる場合もあります。

煙幕を採用したゲームには、例えば「ラリーX」があります。このゲームでは、煙幕に巻き込むことによって、敵を一定時間足止めすることができます。煙幕を使うとエネルギーが減るため、煙幕の使用回数には上限があります。

## アルゴリズム Algorithm

煙幕を実現するには、例えばFig. 6-43のようにします。生成した煙幕は、だんだん大きくなります。最大のサイズに達したら、今度はだんだん小さくなり、最後に消滅します。煙幕のサイズを少しずつ変化させれば、滑らかな動きになります。

**Fig. 6-43** 煙幕の処理

## プログラム Program

List 6-12は煙幕のプログラムです。このプログラムでは、煙幕を出すときに、その位置にすでに煙幕が出ていないかどうかを調べます。複数の煙幕を重ねて出すと、きれいに表示できないためです。煙幕が出ていないときだけ、新しい煙幕を生成します。また、煙幕を生成する位置は、煙幕のサイズに合わせた格子状の座標に揃えています。

**List 6-12** 煙幕(CSmokeクラス、CSmokeManクラス)

```
// 煙幕の移動処理を行うMove関数
bool CSmoke::Move(const CInputState* is) {

    // サイズが変化するスピード
    float vsize=0.1f;

    // 最大のサイズ
```

```
    float max_size=1.0f;

    // 状態に応じて分岐する
    switch (State) {

        // 拡大状態
        case 0:

            // サイズを大きくする
            W+=vsize;
            H+=vsize;

            // 最大のサイズに達したら、
            // 縮小状態に移行する
            if (W>=max_size) State=1;
            break;

        // 縮小状態
        case 1:

            // サイズを小さくする
            W-=vsize*0.2f;
            H-=vsize*0.2f;

            // サイズが0になったら、
            // 煙幕を消去する
            if (W<=0) return false;
            break;
    }

    return true;
}

// キャラクターの移動処理を行うMove関数
bool CSmokeMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 煙幕との当たり判定処理を行うための定数
    // X座標の差分の最大値
    float max_dist=0.6f;

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、キャラクターが画面からはみ出さないように補正する
    X+=VX;
```

**List 6-12**

```
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // ボタンを押したら煙幕を出す
    if (is->Button[0]) {

        // キャラクターの位置に煙幕があるかどうかのフラグ
        bool smoke=false;

        // 煙幕の大きさに合わせて座標を格子状に揃える
        float x=(int)(X+0.5f), y=(int)(Y+0.5f);

        // キャラクターの位置に煙幕があるかどうかを調べる
        for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
            CMover* mover=(CMover*)i.Next();
            if (
                mover->Type==1 &&
                abs(x-mover->X)<max_dist &&
                abs(y-mover->Y)<max_dist
            ) {
                smoke=true;
                break;
            }
        }

        // 煙幕がない場合には、新しい煙幕を生成する
        if (!smoke) new CSmoke(x, y);
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「SMOKE」は煙幕を張るアクションのサンプルです。レバーでキャラクターを左右に動かし、ボタンで煙幕を張ります。移動しながら煙幕を張ることもできます。煙幕は一定時間が経過すると消滅します。

**SMOKE** → p. 398

# 泡

敵に当てると、当たった敵をなかに閉じ込めることができる泡です。敵を閉じ込めた泡を体当たりで割ることによって、敵を完全に倒すことができます。

ボタンを押すと、泡を発射することができます（Fig. 6-44）。泡は水平に飛んでいき、敵に接触すると、接触した敵をなかに包み込みます（Fig. 6-45）。

敵を包み込んだ泡は、敵を閉じ込めたまま、ゆっくりと上昇していきます（Fig. 6-46）。ここでキャラクターをジャンプさせて泡に体当たりさせると、泡を割って敵を完全に倒すことができます（Fig. 6-47）。

泡を採用したゲームには、例えば「バブルボブル」があります。このゲームでは、敵を閉じ込めた泡を複数作っておき、まとめて割ると高得点を得ることができます。また、「足場を作る（→ p. 242）」で紹介したように、泡を足場にしてジャンプすることも可能です。泡の射程距離は、比較的短めに調整されています。

**Fig. 6-44** 泡の発射

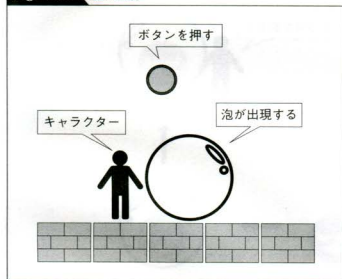



**Fig. 6-45** 敵を包み込む

**Fig. 6-46** 上昇する泡

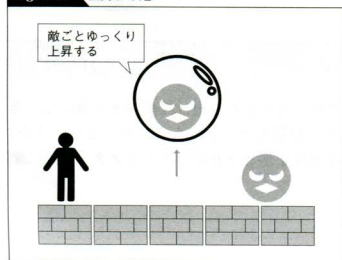



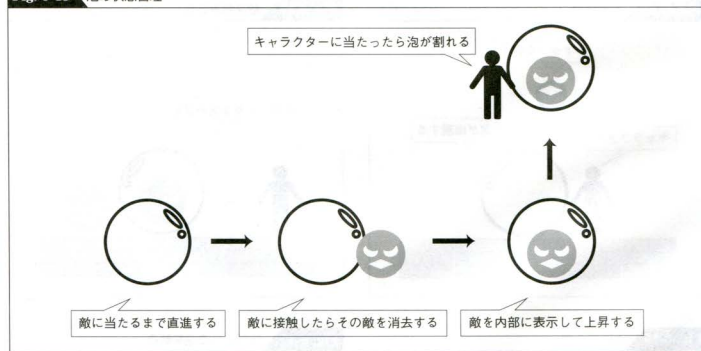
**Fig. 6-47** 体当たりで泡を割る

## アルゴリズム　Algorithm

泡を実現する際のポイントは、泡の状態管理です（Fig. 6-48）。泡には次のような状態があります。

- 発射状態：発射された直後の状態。水平方向に直進する。ゲームによっては、緩やかに上昇する場合もある。敵に接触すると上昇状態に移行する
- 上昇状態：敵を包み込んだ状態。敵をなかに閉じ込めたまま緩やかに上昇する。キャラクターが泡に接触したら破壊状態に移行する
- 破壊状態：キャラクターによって泡が割られたあとの状態。泡が割れるアニメーションなどを表示する。アニメーションが終わったら泡を消去する

接触した敵を閉じ込める処理は、例えば次のように実現します。泡が敵に接触したら、敵を消去します。そして、接触した敵の種類を記録しておき、上昇する泡の内部に、その種類の敵の画像を小さめに表示します。これで、泡に敵が閉じ込められたように見えます。

Fig. 6-48 泡の状態管理

## プログラム　Program

List 6-13は泡のプログラムです。このサンプルでは、発射された泡は画面端まで飛んでいきます。「マシンガン（→ p. 329）」の処理を応用して、射程を短くしてもよいでしょう。また、キャラクターが泡に上から接触したかどうかの判定処理を追加すれば、キャラクターが泡に乗れるようにすることもできます。

**List 6-13** 泡(CBubbleクラス、CBubbleManクラス)

```
// 泡の移動処理を行うMove関数
bool CBubble::Move(const CInputState* is) {

    // 敵との当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=0.6f;

    // 泡の上昇スピード
    float bubble_vy=-0.05f;

    // 泡が小さくなるスピード
    float vsize=0.1f;

    // 状態に応じて分岐する
    switch (State) {

        // 発射状態
        case 0:

            // X座標の更新
            X+=VX;

            // 敵との当たり判定処理
            // 敵に接触したら、敵を消去し、上昇状態に移行する
            // このサンプルでは消去した敵を初期化して、
            // 再び出現させている
            for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
                CMover* mover=(CMover*)i.Next();
                if (
                    mover->Type==2 &&
                    abs(X-mover->X)<max_dist &&
                    abs(Y-mover->Y)<max_dist
                ) {
                    ((CWeaponEnemy*)mover)->Init();
                    State=1;
                }
            }

            // 画面の左右端から出たら、泡を消去する
            if (X<-1 || X>MAX_X) return false;
            break;

        // 上昇状態
        case 1:

            // Y座標の更新
            Y+=bubble_vy;

            // キャラクターとの当たり判定処理
```

**List 6-13**

```
			// キャラクターに接触したら、破壊状態に移行する
			if (
				abs(X-Man->X)<max_dist &&
				abs(Y-Man->Y)<max_dist
			) {
				State=2;
			}

			// 画面の上端から出たら、泡を消去する
			if (Y<-1) return false;
			break;

		// 破壊状態
		case 2:

			// サイズを小さくする
			W-=vsize;
			H-=vsize;

			// サイズが0になったら、泡を消去する
			if (W<=0) return false;
			break;
	}

	return true;
}

// 泡の描画処理を行うDraw関数
void CBubble::Draw() {

	// 泡のサイズを保存する
	float w=W, h=H;

	// 泡が上昇状態か縮小状態のときには、
	// 泡のなかに捕らえられた敵を描画する
	if (State!=0) {

		// 泡のなかに敵を描画する
		// 敵のサイズは泡のサイズに合わせる
		W*=0.7f;
		H*=0.7f;
		Texture=Game->Texture[TEX_ENEMY0];
		CMover::Draw();

		// 泡を描画するために、
		// 画像やサイズを泡用のものに設定する
		Texture=Game->Texture[TEX_BUBBLE];
		W=w;
		H=h;
	}
```

```
    // 泡を描画する
    CMover::Draw();
}

// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 泡のスピード
    float bubble_speed=0.2f;

    // ジャンプの初速度
    float jump_speed=-0.5f;

    // ジャンプ中の加速度
    float jump_accel=0.02f;

    // ジャンプしていないときの処理
    // Jumpはジャンプしているかどうかを表すフラグ
    if (!Jump) {

        // レバーの入力に応じて左右に移動する
        // 攻撃の向きを決めるために、
        // キャラクターが移動した方向を保存しておく
        // VXとVYはキャラクターの速度を表す変数
        // DirXとDirYはキャラクターの移動方向を表す変数
        VX=0;
        if (is->Left) {
            DirX=-1;
            VX=-speed;
        }
        if (is->Right) {
            DirX=1;
            VX=speed;
        }

        // ボタン0を押したら、キャラクターの前方に泡を発射する
        if (!PrevButton && is->Button[0]) {
            new CBubble(X, Y, DirX*bubble_speed, this);
        }

        // ボタン0を押した瞬間を判定するために、
        // 現在のボタンの状態を保存しておく
        PrevButton=is->Button[0];

        // ボタン1を押したらジャンプする
        // ジャンプのフラグとY方向の速度を設定する
        if (is->Button[1]) {
```

**List 6-13**

```
			Jump=true;
			VY=jump_speed;
		}
	} else

	// ジャンプしているときの処理
	{
		// Y方向の速度を更新する
		VY+=jump_accel;

		// Y座標の更新
		Y+=VY;

		// 地面まで落下したら、着地したと判定する
		// Y座標とジャンプのフラグを設定する
		if (VY>0 && Y>=MAX_Y-2) {
			Y=MAX_Y-2;
			Jump=false;
		}
	}

	// X座標を更新し、画面からはみ出さないように補正する
	X+=VX;
	if (X<0) X=0;
	if (X>MAX_X-1) X=MAX_X-1;

	// X方向の速度に応じて、キャラクターを傾けて表示する
	Angle=VX/speed*0.1f;

	return true;
}
```

SAMPLE

「BUBBLE」は泡による攻撃アクションのサンプルです。レバーでキャラクターを左右に動かし、ボタン0で泡を発射します。泡は水平方向に飛んでいき、敵に接触したら、その敵を包んで緩やかに上昇します。ボタン1でキャラクターをジャンプさせて体当たりすれば、泡ごと敵を消すことができます。

**BUBBLE** → p. 398

# 武器切り替え

ボタンで武器を選択するアクションです。キャラクターがいくつもの武器を使い分けるゲームでは、状況に合った武器を素早く選択する楽しみがあります。

武器切り替えを採用したゲームでは、画面の上部や下部などに、なんらかの方法で使用可能な武器の一覧が表示されています（Fig. 6-49）。選択中の武器は、濃い色で描かれるとか、太い枠で囲まれるといった具合に、選択していることが一目でわかるようにされています。

ボタンを押すと、キャラクターは現在選択されている武器を使用します（Fig. 6-50）。ここではボタン0で武器を使うことにしました。

別のボタンを押すと、武器を切り替えることができます（Fig. 6-51）。ここではボタン1で切り替えることにしました。ボタン1を押すたびに、武器一覧において選択中の武器の次の武器に切り替わります。一覧の末尾にある武器まで達したら、先頭の武器に戻ります。

武器を切り替えたあとに再びボタン0を押すと、新たに選択した武器を使います（Fig. 6-52）。武器の種類が多いゲームでは、ボタンを何回押すとどの武器に切り替わるのかを覚えておき、素早く武器を切り替えるといったテクニックが必要です。

武器切り替えを採用したゲームは数多くあります。例えば「最後の忍道」では、刀・手裏剣・爆弾・鎖鎌といった武器を、ボタンで切り替えることができます。また「魂斗羅」シリーズの一部の作品にも、ボタンによる武器切り替えが採用されています。

一方、アイテムを拾うことで武器を切り替えるゲームもあります。例えば「魔界村」や「大魔界村」では、武器を拾うと、その武器に切り替わります。ほかの武器に切り替えるには、また新たに武器を拾う必要があります。「チェルノブ」も同じ方式を採用しています。この方式には、間違った武器を拾ってしまうと、ステージの攻略が大変難しくなるという特徴があります。

**Fig. 6-49** 武器の一覧

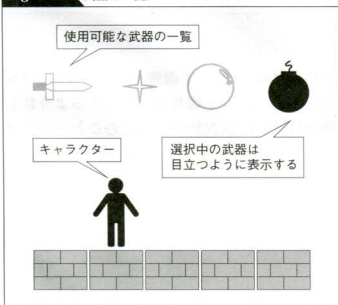



**Fig. 6-50** 武器の使用

**Fig. 6-51** 武器の切り替え

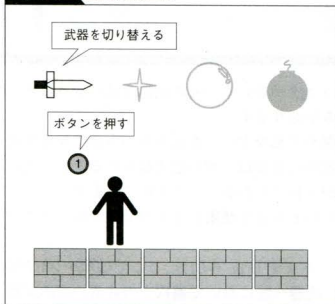
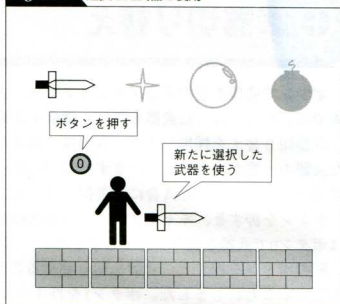



**Fig. 6-52** 選択した武器の使用

「ダイナマイト刑事」もアイテムを拾って武器を切り替えるゲームです。こちらは武器の種類が非常に多く、落ちているもののほとんどが武器として使えます。コショウ・モップ・柱時計、はては桃や寿司船など、およそ武器とは思えないものまで使って攻撃できることが、このゲームの大きな魅力です。

## アルゴリズム Algorithm

武器切り替えを実現するには、武器の一覧を表示する処理と、ボタンを押すたびに武器を切り替える処理が必要です。武器を切り替えるのは、ボタンを押した瞬間だけにしなくてはいけません。ボタンを押したかどうかを判定するだけでは、ボタンを少し押しっぱなしにするだけで、武器が何回も切り替わってしまうからです。

## プログラム Program

List 6-14は武器切り替えのプログラムです。このサンプルでは、「手榴弾（→ p. 317）」「マシンガン（→ p. 329）」「誘導ミサイル（→ p. 332）」「泡（→ p. 349）」を切り替えて使えるようにしました。本章で作成したほかの武器についても、簡単に武器の選択肢に組み込むことができます。

**List 6-14** 武器切り替え（CChangeWeaponManクラス、CChangeWeaponStageクラス）

```
// キャラクターの移動処理を行うMove関数
bool CChangeWeaponMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // 武器の発射スピード
    float weapon_speed=0.2f;

    // 武器の種類の数
    int weapon_count=4;

    // レバーの入力に応じて左右に移動する
    // 攻撃の向きを決めるために、
    // キャラクターが移動した方向を保存しておく
    // VXとVYはキャラクターの速度を表す変数
    // DirXとDirYはキャラクターの移動方向を表す変数
    VX=0;
    if (is->Left) {
        DirX=-1;
        VX=-speed;
    }
    if (is->Right) {
        DirX=1;
        VX=speed;
    }

    // X座標を更新し、画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // ボタン0を押したら、選択している武器を発射する
    // Weaponは選択している武器の番号を表す
    if (!PrevButton[0] && is->Button[0]) {
        switch (Weapon) {

            // 手榴弾
            case 0:
                new CGrenade(X, Y, DirX*weapon_speed, -weapon_speed*2);
                break;

            // マシンガン
            case 1:
                new CMachineGunBullet(X, Y, DirX*weapon_speed);
                break;

            // 誘導ミサイル
```

**List 6-14**

```
			case 2:
				new CHomingMissile(X, Y, -D3DX_PI/2);
				break;

			// 泡
			case 3:
				new CBubble(X, Y, DirX*weapon_speed, this);
				break;
		}
	}

	// ボタン0を押した瞬間を判定するために、
	// 現在のボタン0の状態を保存しておく
	PrevButton[0]=is->Button[0];

	// ボタン1を押したら、次の武器を選択する
	if (!PrevButton[1] && is->Button[1]) {
		Weapon=(Weapon+1)%weapon_count;
	}

	// ボタン1を押した瞬間を判定するために、
	// 現在のボタン1の状態を保存しておく
	PrevButton[1]=is->Button[1];

	// X方向の速度に応じて、キャラクターを傾けて表示する
	Angle=VX/speed*0.1f;

	return true;
}

// ステージの描画処理を行うDraw関数
void CChangeWeaponStage::Draw() {

	// 座標計算の準備
	float sw=Game->GetGraphics()->GetWidth()/MAX_X;
	float sh=Game->GetGraphics()->GetHeight()/MAX_Y;

	// 武器のテクスチャ番号
	int texture[]={TEX_BOMB, TEX_BULLET, TEX_MISSILE, TEX_BUBBLE};

	// 武器の一覧を描画する
	for (int i=0; i<4; i++) {

		// 座標の計算
		float x=sw*(i*3+3), y=sh*2;

		// 色の設定
		// 選択中の武器は濃い色に、
		// 選択していない武器は薄い色にする
		D3DCOLOR color=(Man->Weapon==i?COL_BLACK:COL_LGRAY);
```

```
            // テクスチャの描画
            Game->Texture[texture[i]]->Draw(
                x, y, color, 0, 0,
                x+sw, y, color, 1, 0,
                x, y+sh, color, 0, 1,
                x+sw, y+sh, color, 1, 1
            );
        }
    }
```

SAMPLE

「SWITCHING WEAPON」は武器切り替えのサンプルです。レバーでキャラクターを左右に動かし、ボタン0で武器を切り替えて、ボタン1で武器を発射します。選択ボタンを押すごとに武器が切り替わります。

**SWITCHING WEAPON** → p. 398

## まとめ Stage06

本章では、敵に攻撃するために使う「武器」について解説しました。アクションゲームを遊んでいるときには、武器を効率よく敵に命中させようとしているため、武器の動きは非常によく目に入ります。見ているだけで楽しい動きの武器を組み込めば、ゲームの面白さは間違いなくアップするでしょう。

というわけで、「見ているだけで楽しくなる武器をゲームに組み込もう！」というのが本章のまとめです。

アクションゲームには、さまざまなアイテムが出現します。拾うとダメージを受けるようなアイテムもありますが、ほとんどのアイテムは、拾うとなんらかのメリットがあります。オーソドックスなのは得点が入るアイテムですが、ほかにも無敵になるアイテムや、拾って投げることのできるアイテムなどもあります。また、アイテムの出現方法にもいくつかのパターンがあります。

# アイテム

## Item

ActionGame Algorithm Maniax

Stage

# アイテムで無敵になる

拾うと一定時間キャラクターが無敵になるアイテムです。無敵状態の間は、敵に体当たりしてダメージを与えることができます。

アイテムはキャラクターが接触すると拾うことができます（Fig. 7-1）。アイテムを拾ったキャラクターは、一定時間無敵状態になります（Fig. 7-2）。ここではキャラクターを巨大化させて無敵状態を表現しました。無敵状態では、敵に接触してもダメージを受けません。逆に、接触した敵にダメージを与えることもできます。

無敵になるアイテムを採用したゲームには、例えば「パックマン」があります。このゲームでは、特定のアイテムを食べることによって、キャラクターが一定時間無敵になります。無敵状態では、敵に体当たりすることによって、敵を食べて倒すことができます。

「ちゃっくんぽっぷ」にも無敵アイテムがあります。このゲームでは敵を爆弾で倒しますが、敵を倒したときに無敵アイテムが出ることがあります。これを拾うと、一定時間キャラクターが無敵になり、体当たりで敵を倒せるようになります。

Fig. 7-1 アイテムを拾う

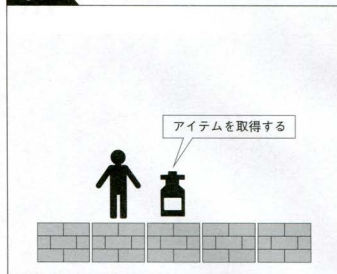
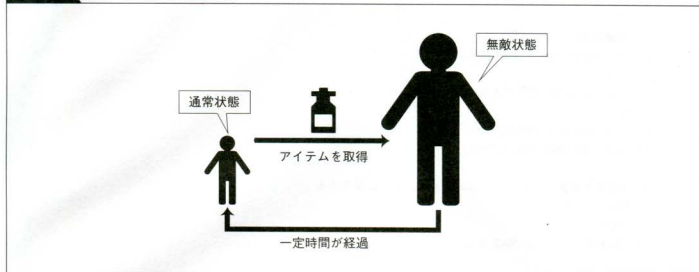



Fig. 7-2 無敵状態

## アルゴリズム Algorithm

アイテムで無敵になるアクションを実現するには、キャラクターの状態管理が必要です（Fig. 7-3）。通常状態のキャラクターがアイテムを取得すると、無敵状態になります。無敵状態で一定時間が経過すると、通常状態に戻ります。

**Fig. 7-3** アイテムで無敵になる処理

## プログラム Program

List 7-1はアイテムで無敵になるアクションのプログラムです。このサンプルでは、キャラクターを大きく表示することによって、無敵状態を表現しました。ほかにも、キャラクターを点滅させるとか、キャラクターの周囲にエフェクトを出すといった方法で、無敵状態を表現してもよいでしょう。

**List 7-1** アイテムで無敵になる(CItemクラス、CInvincibleItemManクラス)

```
// アイテムの移動処理を行うMove関数
bool CItem::Move(const CInputState* is) {

    // 角度の更新
    // アイテムを左右に振動させる
    Angle=sin(Time*0.1f)*0.1f;

    // 時間の更新
    Time++;

    return true;
}

// キャラクターの移動処理を行うMove関数
bool CInvincibleItemMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // アイテムとの当たり判定処理を行うための定数
    // X座標の差分の最大値
```

### List 7-1

```
float max_dist=0.6f;

// 無敵時間（フレーム数）
int invincible_time=180;

// レバーの入力に応じて左右に移動する
VX=0;
if (is->Left) VX=-speed;
if (is->Right) VX=speed;

// X座標を更新し、画面からはみ出さないように補正する
X+=VX;
if (X<0) X=0;
if (X>MAX_X-1) X=MAX_X-1;

// 通常状態の処理
// InvincibleTimeは無敵状態の残り時間を表す
// 残り時間が0のときは、通常状態になる
if (InvincibleTime==0) {

    // アイテムに接触したら、アイテムを拾う
    // 拾ったアイテムは座標を初期化して再出現させる
    // また、無敵時間を設定する
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==2 &&
            abs(X-mover->X)<max_dist &&
            abs(Y-mover->Y)<max_dist
        ) {
            ((CItem*)mover)->Init();
            InvincibleTime=invincible_time;
        }
    }

    // サイズとY座標の設定
    W=H=1;
    Y=MAX_Y-2;

    // 当たり判定の種類を設定する
    Type=0;
} else

// 無敵状態の処理
{
    // 無敵時間を減らす
    InvincibleTime--;

    // サイズとY座標の設定
    // 無敵の雰囲気を表すため、
```

```
        // キャラクターを大きく表示する
        W=H=2;
        Y=MAX_Y-2.5f;

        // 当たり判定の種類を設定する
        // 接触した敵を倒せるようにする
        Type=1;
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「INVINCIBLE ITEM」はアイテムで無敵になるアクションのサンプルです。レバーでキャラクターを左右に動かしてアイテムを拾います。アイテムを拾うとキャラクターは一定時間無敵になり(巨大化し)、体当たりで敵を倒すことができます。

**INVINCIBLE ITEM** → p. 399

# アイテムを拾って投げる

拾ったアイテムを投げるアクションです。投げたアイテムを敵にぶつけると、ダメージを与えることができます。

ここでは爆弾のアイテムを考えましょう (Fig. 7-4)。アイテムに接触すると、キャラクターがアイテムを拾います (Fig. 7-5)。キャラクターはアイテムを持ったまま、自由に移動することができます。

**Fig. 7-4** アイテム

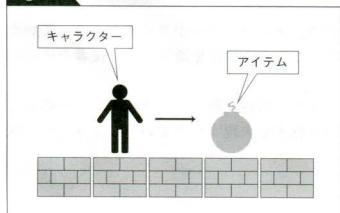



**Fig. 7-5** アイテムを拾う

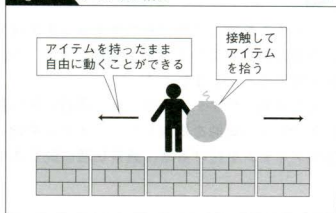



Fig. 7-6 アイテムを投げる

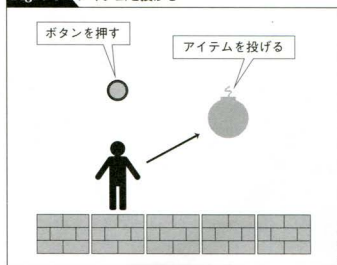
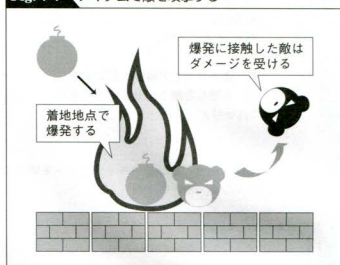



Fig. 7-7 アイテムで敵を攻撃する

ボタンを押すと、手に持っているアイテムを投げます (Fig. 7-6)。爆弾の場合は、投げると放物線を描いて飛んでいき、着地すると爆発します (Fig. 7-7)。爆発に敵を巻き込むと、ダメージを与えることができます。

アイテムを拾って投げるアクションを採用したゲームには、例えば「ぶたさん」があります。このゲームでは、爆弾を拾って投げ、敵に直撃させたり、爆風に巻き込んだりして攻撃します。ボタンを押す長さによって爆弾の飛距離を変えたり、飛んでくる爆弾を伏せて避けたり、敵を殴って持っている爆弾を落とさせたりと、シンプルなルールのなかにもいろいろな要素が盛り込まれています。

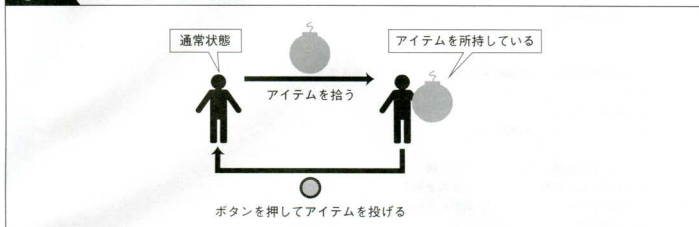
「ダイナマイト刑事」では、ステージに落ちている多種多様なアイテムを拾って投げることができます。「武器切り替え (→ p. 355)」でも紹介したように、アイテムのなかには投げて使うものもあれば、振り回して使うものもあります。

「ぺんぎんくんウォーズ」は、アイテムを投げるアクションを採用したゲームの少し変わった例です。ボールを拾って投げることによって、すべてのボールを敵側のフィールドに送り込めば勝ちになります。ボールを敵に当てると気絶させることができるので、そのすきにボールを送り込むのが有効な戦術です。

## アルゴリズム Algorithm

アイテムを拾って投げるアクションを実現するには、キャラクターがアイテムを持っていない状態と、持っている状態を区別します (Fig. 7-8)。ここでは前者を通常状態、後者を所持状態と呼ぶことにしましょう。

通常状態でアイテムを拾うと、所持状態に移行します。所持状態では、キャラクターの近くにアイテムの画像を描画して、アイテムを持っている様子を表現するとよいでしょう。所持状態でアイテムを投げると、通常状態に戻ります。

Fig. 7-8 アイテムを拾って投げる処理

## プログラム Program

List 7-2はアイテムを拾って投げるアクションのプログラムです。このサンプルでは「手榴弾(→ p. 317)」を拾って投げます。ほかのアイテムを拾って投げる処理も、同じ要領で実現することができます。

**List 7-2** アイテムを拾って投げる(CPickAndThrowItemManクラス)

```
// キャラクターの移動処理を行うMove関数
bool Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // アイテムを投げるスピード
    float throw_speed=0.2f;

    // アイテムとの当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=0.6f;

    // レバーの入力に応じて左右に移動する
    // アイテムを投げる向きを決めるために、
    // キャラクターが移動した方向を保存しておく
    // VXとVYはキャラクターの速度を表す変数
    // DirXとDirYはキャラクターの移動方向を表す変数
    VX=0;
    if (is->Left) {
        DirX=-1;
        VX=-speed;
    }
    if (is->Right) {
```

**List 7-2**

```
		DirX=1;
		VX=speed;
	}

	// X座標を更新し、画面からはみ出さないように補正する
	X+=VX;
	if (X<0) X=0;
	if (X>MAX_X-1) X=MAX_X-1;

	// アイテムを持っていないときの処理
	// アイテムに接触したら、アイテムを拾う
	// Itemはアイテムを持っているかどうかのフラグ
	if (!Item) {
		for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
			CMover* mover=(CMover*)i.Next();
			if (
				mover->Type==2 &&
				abs(X-mover->X)<max_dist &&
				abs(Y-mover->Y)<max_dist
			) {
				((CItem*)mover)->Init();
				Item=true;
			}
		}
	} else

	// アイテムを持っているときの処理
	// ボタンを押したら、キャラクターの前方斜め上にアイテムを投げる
	// アイテムを投げたら、アイテムを持っていない状態に戻る
	{
		if (!PrevButton && is->Button[0]) {
			new CGrenade(X, Y-1, DirX*throw_speed, -throw_speed*2);
			Item=false;
		}
	}

	// ボタンを押した瞬間を判定するために、
	// 現在のボタンの状態を保存しておく
	PrevButton=is->Button[0];

	// X方向の速度に応じて、キャラクターを傾けて表示する
	Angle=VX/speed*0.1f;

	return true;
}
```

```
// キャラクターの描画処理を行うDraw関数
void Draw() {

    // 角度を保存する
    float angle=Angle;

    // アイテムを持っているときには、
    // キャラクターの頭上にアイテムを描画する
    if (Item) {
        Texture=Game->Texture[TEX_BOMB];
        Y=MAX_Y-3;
        Angle=0;
        CMover::Draw();
    }

    // キャラクターの本体を描画する
    Texture=Game->Texture[TEX_MAN];
    Y=MAX_Y-2;
    Angle=angle;
    CMover::Draw();
}
```

**SAMPLE**

「THROWING」はアイテムを拾って投げるアクションのサンプルです。レバーでキャラクターを左右に動かしてアイテムを拾います。拾ったアイテムは、ボタンで投げることができます。

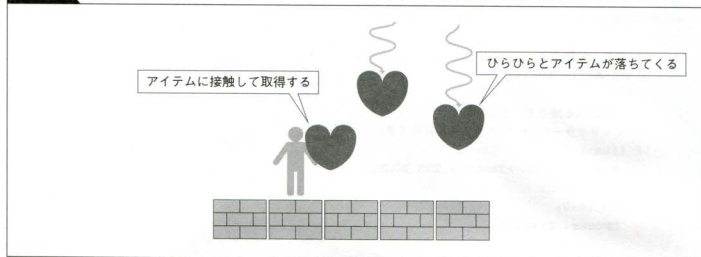
**THROWING → p. 399**

## 舞い落ちるアイテム

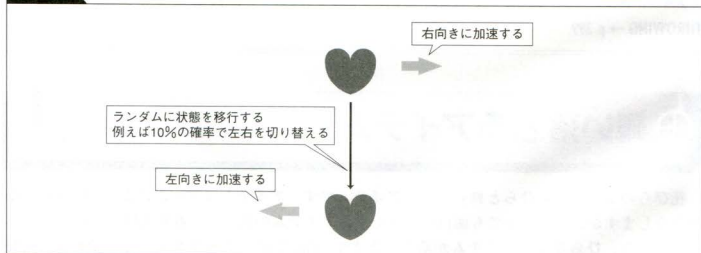
花びらのようにひらひらと舞い落ちるアイテムです。アイテムはゲームによっていろいろな動きをしますが、そのなかでも面白い動きをするアイテムの例として取り上げました。

上空から、ひらひらとアイテムが落ちてきます(Fig. 7-9)。キャラクターをアイテムに接触させると、アイテムを拾うことができます。アイテムは左右にひらひらと動くので、上手に追いかけないと拾うことができません。

舞い落ちるアイテムを採用したゲームには、例えば「ポパイ」があります。このゲームでは、ひらひらと降ってくるハートや音符を拾うことが目的です。また、関連作品の「ポパイの英語遊び」には、ひらひらと舞い落ちるアルファベットを拾って、単語を作るゲームがあります。

Fig. 7-9 舞い落ちるアイテム

## アルゴリズム Algorithm

舞い落ちるアイテムのポイントは、ひらひらとしたアイテムの動きです(Fig. 7-10)。これは、アイテムを右向きに加速する状態と、左向きに加速する状態を、ランダムに移行することで実現できます。低い確率で両方の状態を移行するようにすれば、アイテムは現実の花びらや落ち葉のように、左右へひらひらと動きます。

Fig. 7-10 舞い落ちるアイテムの処理

## プログラム Program

List 7-3は舞い落ちるアイテムのプログラムです。左右方向への加速度や、状態を移行する確率を変えると、アイテムの動きが変わります。パラメータを調整して、見た目に楽しい動きになるようにするとよいでしょう。また、動きに合わせて画像の回転表示などを行うと、さらに見た目が面白くなります。

**List 7-3** 舞い落ちるアイテム(CFallingItemクラス、CFallingItemManクラス)

```
// アイテムの移動処理を行うMove関数
bool CFallingItem::Move(const CInputState* is) {

    // 左右方向への加速度
    float accel=0.002f;

    // 低い確率で左右の進行方向を逆転させる
    if (Game->Rand.Real1()<0.1f) DirX=-DirX;

    // 進行方向に加速させる
    VX+=accel*DirX;

    // X座標とY座標の更新
    X+=VX;
    Y+=VY;

    // 画面の下端から出たら、
    // 位置を初期化して、画面の上端から再出現させる
    if (Y>=MAX_Y) Init();

    return true;
}

// キャラクターの移動処理を行うMove関数
bool CFallingItemMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // アイテムとの当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=0.6f;

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // アイテムとの当たり判定処理
    // アイテムに接触したら、アイテムを消去する
    // このプログラムでは、消去したアイテムの位置を初期化して、
    // 画面の上端から再び出現させる
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
```

**List 7-3**

```
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==2 &&
            abs(X-mover->X)<max_dist &&
            abs(Y-mover->Y)<max_dist
        ) {
            ((CFallingItem*)mover)->Init();
        }
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「FALLING ITEM」は舞い落ちるアイテムのサンプルです。画面上部からひらひらとアイテムが舞い落ちてきます。レバーでキャラクターを左右に移動させて、アイテムを拾うことができます。

**FALLING ITEM** → p. 399

## 特定の場所を通るとアイテム出現

キャラクターが特定の場所を通過すると、アイテムが出現するアクションです。どこを通過するとアイテムが出現するのかは、明示されている場合もあれば、隠されている場合もあります。

例えば、キャラクターが特定の場所を通過したとします（Fig. 7-11）。すると、アイテムが出現します（Fig. 7-12）。アイテムは通過した場所に出現することもあれば、少し離れた場所に出現することもあります。

特定の場所を通るとアイテムが出現するゲームの例としては、「クルクルランド」があります。このゲームでは、金塊が隠された通路をキャラクターが通過すると、金塊が出現します。ステージにあるすべての金塊を出現させることがゲームの目的です。金塊の配列は絵や模様になっているので、金塊を出現させるにつれて、だんだんどんな絵なのかがわかっていく楽しみがあります。

「パックランド」では、特定の場所を通過したり、消火栓などの仕掛けに乗ったりすると、得点アイテムが出現します。得点アイテムを連続して取ると高得点が得られるので、アイテムの出現位置を覚えておけば、効率よく回収することができます。

Fig. 7-11 特定の場所を通過する

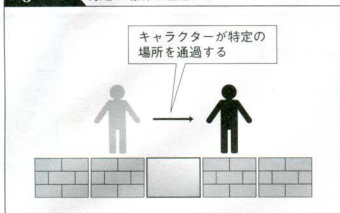
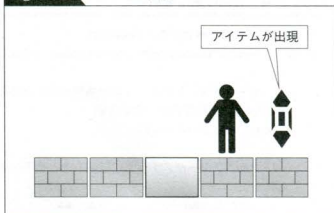
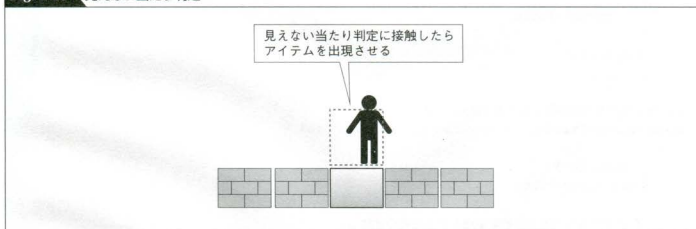



Fig. 7-12 アイテムの出現

## アルゴリズム Algorithm

特定の場所を通ったときにアイテムを出現させるには、見えない当たり判定を使います(Fig. 7-13)。これはいわば、アイテムを出現させるためのスイッチです。

当たり判定にキャラクターが接触したら、アイテムを出現させます。通過すると何かが起こるということをプレイヤーに伝えたいときには、当たり判定の下にある床を特別な形にしたり、当たり判定のある場所にエフェクトを表示したりするとよいでしょう。

Fig. 7-13 見えない当たり判定

## プログラム Program

List 7-4は特定の場所を通ったときにアイテムを出現させるプログラムです。見えないスイッチの移動処理では、キャラクターとの当たり判定処理を行い、キャラクターが接触したときにはアイテムを出現させます。キャラクターが1回通過するごとにアイテムを1個だけ出現させるために、フラグを用いてキャラクターがスイッチに接触した瞬間を判定しています。

**List 7-4** 特定の場所を通るとアイテム出現(CItemByPassSwitchクラス、CItemByPassManクラス)

```
// スイッチの移動処理を行うMove関数
bool CItemByPassSwitch::Move(const CInputState* is) {

    // キャラクターとの当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=0.6f;

    // キャラクターがスイッチに接触したときの処理
    if (
        abs(X-Man->X)<max_dist &&
        abs(Y-Man->Y)<max_dist
    ) {
        // キャラクターがスイッチに接触した瞬間ならば、
        // アイテムを出現させる
        if (!PrevHit) new CItem(Game->Texture[TEX_JEWEL]);

        // スイッチに接触したフラグをtrueにする
        // このフラグはスイッチに接触した瞬間を判定するために使う
        PrevHit=true;
    } else

    // キャラクターがスイッチに接触していないときの処理
    // スイッチに接触したフラグをfalseにする
    {
        PrevHit=false;
    }
    return true;
}

// キャラクターの移動処理を行うMove関数
bool CItemByPassMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // アイテムとの当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float max_dist=0.6f;

    // レバーの入力に応じて左右に移動する
    VX=0;
    if (is->Left) VX=-speed;
    if (is->Right) VX=speed;

    // X座標を更新し、画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;
```

```
    // アイテムとの当たり判定処理
    // アイテムに接触したら、アイテムを消去する
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==2 &&
            abs(X-mover->X)<max_dist &&
            abs(Y-mover->Y)<max_dist
        ) {
            i.Remove();
        }
    }

    // X方向の速度に応じて、キャラクターを傾けて表示する
    Angle=VX/speed*0.1f;

    return true;
}
```

SAMPLE

「ITEM BY PASS」は特定の場所を通るとアイテムが出現するアクションのサンプルです。レバーでキャラクターを左右に動かして、画面中央のスイッチの上を通過すると、アイテムが1つ出現します。

**ITEM BY PASS** → p. 399

## 特定の場所を攻撃するとアイテム出現

特定の場所に攻撃を当てると、アイテムが出現するアクションです。例えば、特定のブロックに体当たりしたり、特定の壁を武器で撃ったりすると、アイテムが出現します。

ここでは、ジャンプしてブロックに体当たりすることを考えましょう (Fig. 7-14)。攻撃されたブロックは振動します。それと同時に、ブロックの上にアイテムが出現します (Fig. 7-15)。ジャンプしてアイテムに接触すると、アイテムを取ることができます。

特定の場所を攻撃したときにアイテムが出現するゲームには、例えば「スーパーマリオブラザーズ」があります。このゲームでは、ジャンプしてブロックを下から突き上げると、コインやキノコといったアイテムが出現します。数多くのアイテムが隠されているので、あちこちのブロックに攻撃して、アイテムを探す楽しみがあります。

Fig. 7-14 ブロックへの攻撃

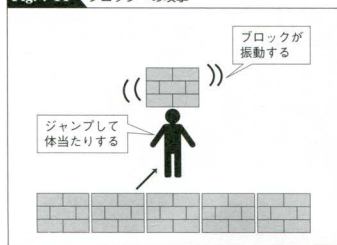
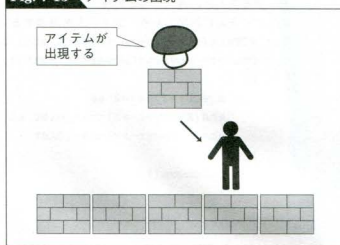
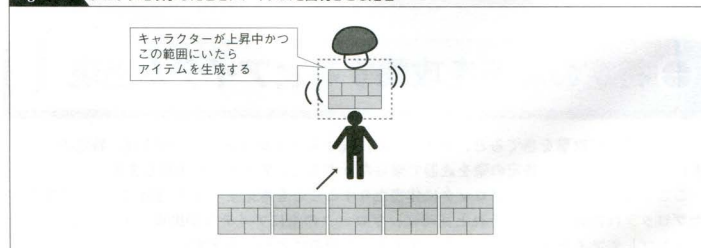



Fig. 7-15 アイテムの出現

## アルゴリズム Algorithm

特定の場所を攻撃したときにアイテムを出現させるには、キャラクターと攻撃される場所（アイテムが隠されている場所）の当たり判定処理を行います。例えば、ブロックを突き上げる場合には、上昇中のキャラクターがブロックに接触したかどうかを調べます。もし接触したら、アイテムを出現させます（Fig. 7-16）。

Fig. 7-16 ブロックを攻撃したときにアイテムを出現させる処理

## プログラム Program

List 7-5は、特定の場所を攻撃したときにアイテムを出現させるプログラムです。このサンプルでは、ブロックをジャンプで突き上げたときに、ブロックの上にアイテムが出現します。ブロックとアイテムは、突き上げられた雰囲気が出るように、少し跳ね上がってから元の位置に落ちてくる動きにしています。

**List 7-5** 特定の場所を攻撃するとアイテム出現(CItemByAttackItemクラス、CItemByAttackBlockクラス、CItemByAttackManクラス)

```
// アイテムの移動処理を行うMove関数
bool CItemByAttackItem::Move(const CInputState* is) {

    // 落下の加速度
    float accel=0.02f;

    // Y方向の速度を更新する
    VY+=accel;

    // Y座標の更新
    Y+=VY;

    // 跳ね上がったアイテムを着地させる処理
    // Y座標が初期値を超えたら、
    // Y座標を初期値に戻し、Y方向の速度を0にする
    if (Y>InitialY) {
        Y=InitialY;
        VY=0;
    }

    return true;
}

// ブロックを攻撃するときに呼び出すAttack関数
void CItemByAttackBlock::Attack() {

    // ブロックを跳ね上げるために、上向きの初速度を設定する
    VY=-0.1f;

    // ブロックの上にアイテムを生成する
    new CItemByAttackItem(X, Y-1);
}

// ブロックの移動処理を行うMove関数
bool CItemByAttackBlock::Move(const CInputState* is) {

    // 落下の加速度
    float accel=0.02f;

    // Y方向の速度を更新する
    VY+=accel;

    // Y座標の更新
    Y+=VY;

    // 跳ね上がったブロックを着地させる処理
    // Y座標が初期値を超えたら、
    // Y座標を初期値に戻し、Y方向の速度を0にする
```

**List 7-5**

```
    if (Y>InitialY) {
        Y=InitialY;
        VY=0;
    }

    return true;
}

// キャラクターの移動処理を行うMove関数
bool CItemByAttackMan::Move(const CInputState* is) {

    // 移動スピード
    float speed=0.2f;

    // ジャンプの初速度
    float jump_speed=-0.5f;

    // ジャンプ中の加速度
    float jump_accel=0.02f;

    // 床またはブロックとの当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float floor_dist=1.0f;

    // アイテムとの当たり判定処理を行うための定数
    // X座標とY座標の差分の最大値
    float item_dist=0.6f;

    // ジャンプしていないときの処理
    // Jumpはジャンプしているかどうかを表すフラグ
    if (!Jump) {

        // レバーの入力に応じて左右に移動する
        VX=0;
        if (is->Left) VX=-speed;
        if (is->Right) VX=speed;

        // ボタンを押したらジャンプする
        if (is->Button[0]) VY=jump_speed;
    }

    // X座標を更新し、画面からはみ出さないように補正する
    X+=VX;
    if (X<0) X=0;
    if (X>MAX_X-1) X=MAX_X-1;

    // 床またはブロックとの当たり判定処理(左右方向)
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
```

```
            mover->Type==1 &&
            abs(X-mover->X)<floor_dist &&
            abs(Y-mover->Y)<floor_dist
        ) {
            // ブロックに右から接触したら、
            // キャラクターをブロックの右側に密着させる
            if (VX<0) {
                X=mover->X+floor_dist;
            } else

            // ブロックに左から接触したら、
            // キャラクターをブロックの左側に密着させる
            {
                X=mover->X-floor_dist;
            }
        }
    }

    // Y方向の速度を更新する
    VY+=jump_accel;

    // Y座標の更新
    Y+=VY;

    // 床またはブロックとの当たり判定処理(上下方向)
    Jump=true;
    for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();
        if (
            mover->Type==1 &&
            abs(X-mover->X)<floor_dist &&
            abs(Y-mover->Y)<floor_dist
        ) {
            // ブロックに下から接触したら、
            // キャラクターをブロックの下側に密着させる
            // また、ブロックを上に跳ね上げる
            if (VY<0) {
                ((CItemByAttackBlock*)mover)->Attack();
                Y=mover->Y+floor_dist;
            } else

            // 床またはブロックに上から接触したら着地する
            {
                Jump=false;
                Y=mover->Y-floor_dist;
            }

            // Y方向の速度を0にする
            VY=0;
        }
```

**List 7-5**

```
	}

	// アイテムとの当たり判定処理
	// アイテムに接触したら、アイテムを消去する
	for (CTaskIter i(Game->MoverList); i.HasNext(); ) {
		CMover* mover=(CMover*)i.Next();
		if (
			mover->Type==2 &&
			abs(X-mover->X)<item_dist &&
			abs(Y-mover->Y)<item_dist
		) {
			i.Remove();
		}
	}

	// X方向の速度に応じて、キャラクターを傾けて表示する
	Angle=VX/speed*0.1f;

	return true;
}
```

SAMPLE

「ITEM BY ATTACK」は特定の場所を攻撃するとアイテムが出現するアクションのサンプルです。レバーでキャラクターを左右に動かして、ボタンでジャンプします。ジャンプでブロックを突き上げると、ブロックの上にアイテム(キノコ)が出現します。

**ITEM BY ATTACK** → p. 399

## まとめ Stage07

本章では、拾うと得点が入ったり、さまざまな効果が得られたりする「アイテム」について解説しました。アイテムでゲームを面白くするには、アイテムの効果や見た目を工夫することはもちろん、アイテムを出現させる方法や、アイテムを回収する方法もよく検討する必要があります。

というわけで、「アイテム自体を面白くするだけではなく、出現方法や回収方法も楽しめるようにしよう！」というのが本章のまとめです。

本書ではここまでに、アクションゲームで使える数多くの要素を紹介してきました。これらの要素はアクションゲーム制作に利用することができますが、実際にゲームを作るには、ゲームの核となるものが必要です。それはミッション、つまりゲームの目的です。本章では、敵を倒したり、アイテムを取ったりといった、プレイヤーに課せられるミッションのいろいろなパターンを紹介します。

# ミッション

## Mission

ActionGame Algorithm Maniax

# Stage 08

# すべての敵を倒す

ステージにいる敵をすべて倒すミッションです。ステージには多くの敵がいます（Fig. 8-1）。キャラクターを動かしたり武器を使ったりして、すべての敵を倒したらステージクリアとなります（Fig. 8-2）。

ゲームによっては、小さな敵を全滅させたうえで、ステージの最後に登場するボスを倒したらクリア、というパターンもあります。あるいは、小さな敵は倒しても倒さなくてもかまわなくて、最後のボスだけを倒したらクリアという場合もあります。

すべての敵を倒すミッションを採用したゲームには、例えば「平安京エイリアン」があります。このゲームでは、ステージにいる数匹の敵をすべて埋めて倒すと、ステージクリアとなります。

「ゴールデンアックス」では、スクロールするステージを進んでいきますが、スクロールがときどき止まります。そのときに画面内にいる敵を全滅させると、スクロールが再開して、ステージの先に進むことができます。これもすべての敵を倒すミッションの一種です。

## アルゴリズム　Algorithm

すべての敵を倒すミッションを実現するには、ステージ内にいる敵の数をカウントします。敵を倒すたびにカウントを減らしていき、カウントが0になったらステージクリアとします。

**Fig. 8-1** ステージ内に配置された敵

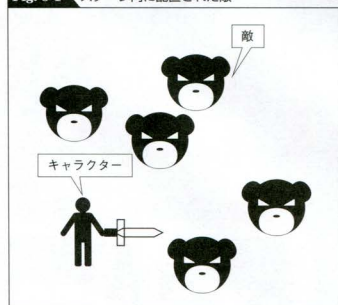
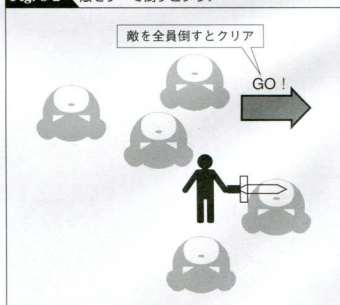



**Fig. 8-2** 敵をすべて倒すとクリア

# すべてのアイテムを取る

ステージ内にあるアイテムをすべて取るミッションです。ステージ内には多くのアイテムが配置されています (Fig. 8-3)。キャラクターを移動させて、アイテムをすべて拾うとステージクリアとなります (Fig. 8-4)。

ゲームによっては、すべてのアイテムを拾ったあとに、特定の場所にゴールが出現する場合もあります。また、多くのゲームでは、ジャンプや仕掛けを駆使しないと取れない場所にアイテムがあったり、アイテムを取るのを敵がじゃましたりして、一筋縄ではアイテムが回収できないようにしてあります。

すべてのアイテムを取るミッションを採用したゲームは数多くあります。例えば「マッピー」では、家のなかに置かれた品物をすべて回収するとステージクリアとなります。追いかけてくる敵は、「トランポリン (→ p. 133)」や「ドア飛ばし (→ p. 148)」を使ってかわします。また、品物を取る順番を工夫して、同じ品物のペアを続けて回収すると、高得点が得られます。

「ラリーX」では、スクロールするステージに配置された旗を回収します。追撃してくる敵は、「煙幕 (→ p. 345)」を使って足どめします。

「パックマン」は、アイテムを回収するゲームの代表格です。ステージ内に配置された数多くのドット (点) を、すべて食べるとクリアとなります。ドットはステージ内にくまなく配置されているので、「ペイント (→ p. 385)」ミッションの一種と見ることもできます。追撃してくる敵は、「アイテムで無敵になる (→ p. 362)」アクションを使って倒します。なお、「パックマン」のように画面上のドットをすべて消すゲームの形式を「ドットイーター」と呼ぶことがあります。

「ロードランナー」も、ステージ内のアイテムを回収するタイプのゲームです。このゲームでは「自動穴 (→ p. 216)」を使って敵を足どめしたり、敵を埋めて倒したりしながら、金塊を回収します。ステージによっては、上手に穴を掘って深いところにある金塊を回収したり、「ロープ (→ p. 124)」や「はしご (→ p. 128)」を利用して巧みに敵から逃れたりする必要があります。アイテム回収ゲームのなかでも、特にパズル要素が強いゲームです。

**Fig. 8-3** ステージ内に配置されたアイテム

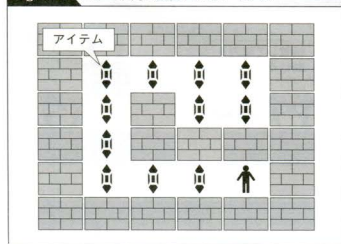
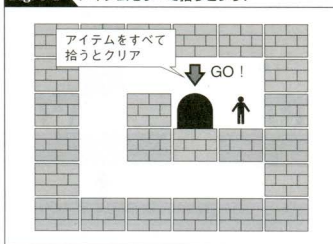



**Fig. 8-4** アイテムをすべて拾うとクリア

「レッキングクルー」では、ステージにあるすべての壁を壊すことが目的ですが、内容としてはアイテムを回収するのに似ています。硬い壁は何度も叩かなければ壊れません。また、追ってくる敵につかまらないように、敵のすきを突いて壁を叩く必要があります。

「新入社員とおる君」にも、アイテムを回収するステージがあります。ステージ内にあるハートを回収するのですが、そのためには椅子に座っているほかの社員を攻撃して、椅子から突き落とす必要があります。敵のすきを突いて社員を攻撃する動きは、「レッキングクルー」に似ています。

## アルゴリズム Algorithm

すべてのアイテムを取るミッションを実現するには、ステージ内にあるアイテムの数をカウントします。アイテムを取るたびにカウントを減らしていき、カウントが0になったらステージクリアとします。カウントが0になった時点で出口を出現させて、キャラクターが出口に入ったらクリア、としているゲームもあります。

# 味方を助ける

ステージ内にいる味方をすべて助けるミッションです。ステージ内には多くの味方が配置されています (Fig. 8-5)。キャラクターを移動させて、味方に接触すると (Fig. 8-6)、味方がキャラクターについてきます (Fig. 8-7)。

味方を引き連れてゴールに入ると、味方を助け出すことができます (Fig. 8-8)。味方を全員助けたらステージクリアです。ゲームによっては、味方に接触した時点で救出したことになるものもあります。

味方を助けるミッションを採用したゲームには、例えば「フリッキー」があります。このゲームでは、一度に多くの味方を助けるほど、高得点が得られます。味方はキャラクターのあと

**Fig. 8-5** ステージ内に配置された味方

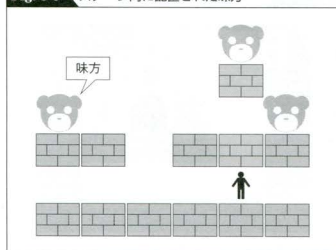



**Fig. 8-6** 味方に接触する

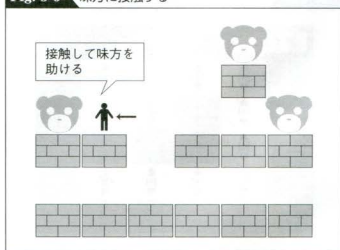



Fig. 8-7 味方がついてくる

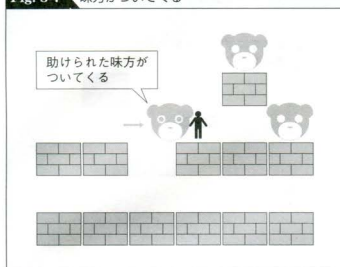
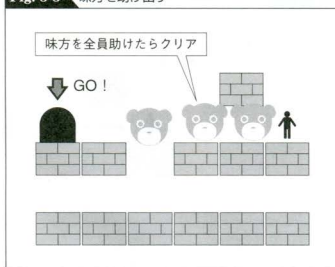



Fig. 8-8 味方を助け出す

をついてくるため、多くの味方を引き連れていると、味方が美しい軌跡を描きます。しかし、連れている味方が敵に触れてしまうと、キャラクターから切り離されてしまいます。味方の列が長くなるほど敵に接触しやすくなるので、上手に敵を避けながらいかに列を長くするかが、このゲームの面白いところです。

## アルゴリズム Algorithm

味方を助けるミッションを実現するには、ステージ内にいる味方の数をカウントしておき、味方を助け出すたびにカウントを減らします。カウントが0になったらステージクリアです。

もう1つのポイントは、味方をキャラクターに追随させる処理です。これはキャラクターが通った座標を記録しておき、同じ座標を味方が少し遅れてたどるようにすれば実現できます。キャラクターの動きに加速度を使うと、味方の軌跡が滑らかな曲線になって、見た目が楽しくなります。

## ペイント

ステージを塗りつぶすミッションです。キャラクターが通過すると、ステージ内のブロックや床などが塗りつぶされます(Fig. 8-9)。すべてのブロックや床を通過して、ステージ全体を塗りつぶしたらクリアとなります(Fig. 8-10)。このミッションは、「パックマン」のような「ドットイーター」形式の一種と見ることもできます。

ペイントを採用したゲームには、例えば「シティコネクション」があります。このゲームでは車を操作して、ジャンプを駆使しながら、段差のあるステージの床を塗りつぶします。すべての床を塗りつぶすとクリアです。パトカーが追いかけてきたり、進路をじゃまする猫が出たりするので、上手にかわしながらペイントを行う必要があります。

**Fig. 8-9** ステージを塗りつぶす

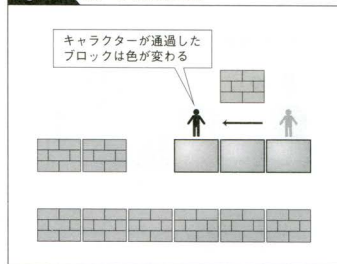
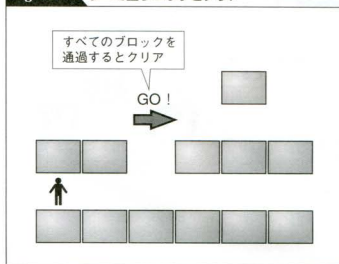



**Fig. 8-10** すべて塗りつぶすとクリア

「キューバート」もペイントを採用したゲームです。キャラクターは立体感のあるステージを跳びはねて移動します。床を通過すると塗りつぶすことができ、すべての床を塗りつぶすとステージクリアです。

### アルゴリズム Algorithm

ペイントを実現するには、ステージ内のブロックや床の数をカウントします。塗るたびにカウントを減らし、カウントが0になったらステージクリアです。同時に、通過したブロックや床は、通過したことがわかるグラフィックに変えていきましょう。

## 目的地へいく

キャラクターを目的地まで移動させるミッションです。多くのゲームでは、敵の攻撃や障害物など、キャラクターが目的地に向かうのをじゃまする要素が用意されています (Fig. 8-11)。ジャンプや特殊行動などを駆使して敵や障害物をかわし、目的地に到達するとステージクリアです。

目的地へいくミッションを採用したゲームには、例えば「フロッガー」があります。このゲームではカエルを操作して、車を避けたり、丸太に飛び移ったりしながら、ゴールの巣を目指します。巣は複数用意されていて、すべての巣にカエルを到達させるとステージクリアになります。

「ゲイングランド」は、目的地へいくミッションと、敵を全滅させるミッションを組み合わせたゲームです。どちらのミッションを達成しても、ステージをクリアすることができます。このゲームには、キャラクターが何種類もあり、それぞれ特性が違います。ステージの特性と

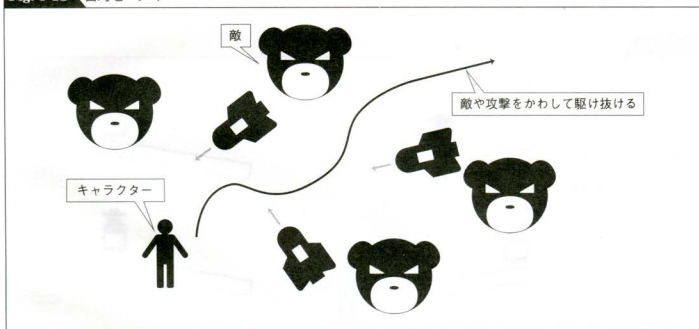
キャラクターの特性を考慮して、目的地へいくのか、敵を全滅させるのかを判断する必要があります。

目的地へいくミッションの変わり種としては、「ルナーランダー」のような着陸ゲームがあります。このゲームでは、月着陸船を目的地に着陸させると、ステージクリアとなります。着陸船の操作は難しく、繊細な操作をしないと、スピード超過で地面に激突してしまいます。敵は出現しませんが、キャラクターの操作だけで十分に難しいゲームです。

狭く入り組んだ通路を通り抜けて、キャラクターをゴールまで到達させるゲームも、目的地へいくミッションの一種だといえます。例えば「クレイジーバルーン」では、左右に揺れ動く風船を操作して、ステージの壁にぶつからないようにゴールを目指します。

また「マーブルマッドネス」は、起伏のあるステージのなかで玉を転がすゲームです。敵に触ったり崖に落ちたりしないように、ゴールまで玉を運びます。立体感のあるグラフィックと、トラックボールを使った独特の操作感が、このゲームの魅力です。

アイテムを拾ってから目的地にいく、というパターンのゲームもあります。例えば「フリスキートム」では、水道管のパーツを拾ってから壊れた箇所にいくと、水道管を直すことができます。

**Fig. 8-11** 目的地へいく

## アルゴリズム Algorithm

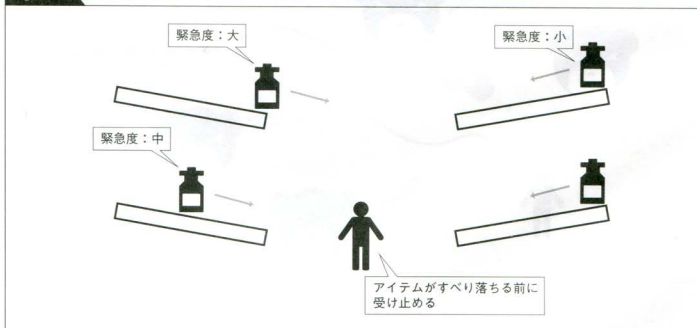
目的地にいくミッションを実現するには、目的地とキャラクターの当たり判定処理を行い、キャラクターが目的地に到達したかどうかを判定します。ゲームデザインとしては、キャラクターの進路をふさぐ敵や障害物をうまく配置できるかどうかが、ゲームの面白さを決めます。

# 待ち行列の処理

次々と迫ってくるアイテムやキャラクターなどの行列を、短時間で素早くさばいていくミッションです。例えば、次々に出現するアイテムを、床にすべり落ちる前に拾うゲームなどがあります(Fig. 8-12)。端に近いアイテムは、端から遠いアイテムよりも早く拾う必要があります。緊急度の大小を見極めて、より緊急に対応する必要があるアイテムから処理していくのが、このミッションのポイントです。

待ち行列の処理を採用したゲームには、例えば「ミッキーマウス」があります。このゲームでは、玉子が4つの坂を転がり落ちてきます。玉子が床に落ちるとミスになるので、落ちる前に拾う必要があります。玉子が落ちてくる順番をよく見ておいて、先に落ちてきた玉子を優先して拾うことが大事です。

「タッパー」も待ち行列の処理を採用したゲームの一種です。このゲームでは、カウンターの端から押し寄せてくる客に対して、素早くビールを配ることが目的です。ビールを飲んだ客が投げてくる空いたジョッキも、落とさないように受け取る必要があります。

Fig. 8-12 待ち行列の処理

## アルゴリズム

待ち行列の処理は、アイテムを拾うアクションなどの組み合わせなので、プログラムとしては特に難しいことはありません。あとは、魅力的なゲームの題材を考え出せるかどうかがポイントです。既存のゲームには、「タッパー」のように飲食店を題材にしたゲームが多いようです。

# ものを撃ち合う

敵や別のプレイヤーとの間でものを撃ち合い（打ち合い）、相手の陣地にものを落とせば勝ち、というミッションです。例えば、ラケットを動かしてボールを打ち合う、テニスのようなゲームがこれに相当します（Fig. 8-13）。ラケットはレバーで上下に動かすことができます。ラケットにボールを当てると跳ね返すことができますが、当て損なって落としてしまうとミスになります（Fig. 8-14）。

別の題材としては、空き缶を撃ち合うゲームもあります（Fig. 8-15）。落下する空き缶に弾丸を当てると、跳ね上げることができます（Fig. 8-16）。弾丸を当てる位置によって、空き缶の動きが微妙に変わります。自分の陣地に空き缶を落とされたら、落とされたプレイヤーのミスとなります。

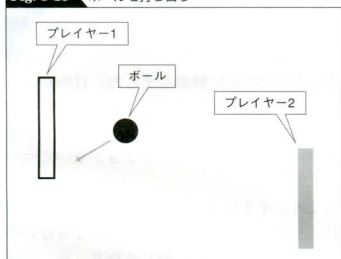
**Fig. 8-13** ボールを打ち合う

プレイヤー1
ボール
プレイヤー2

**Fig. 8-14** ボールを落としたプレイヤーの負け

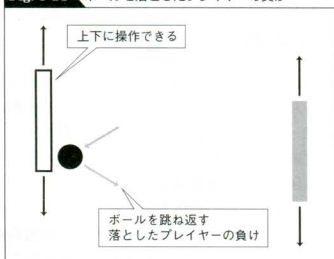



**Fig. 8-15** 空き缶を撃ち合う

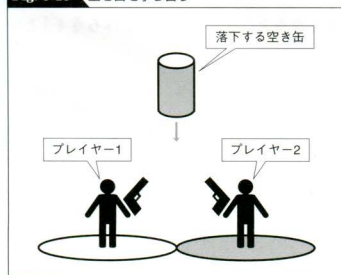
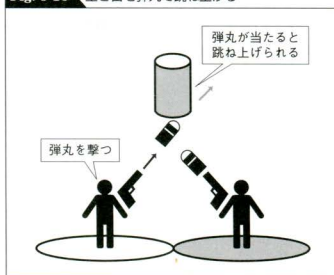



**Fig. 8-16** 空き缶を弾丸で跳ね上げる

「Pong」はテニスのようなゲームの例です。このゲームではテニスや卓球のように、2人のプレイヤーがボールを打ち合います。相手にボールを落とさせると得点が入り、先に一定の得点を稼いだプレイヤーが勝者となります。

「Dog Patch」は空き缶を撃ち合うゲームです。2人のプレイヤーで空き缶を撃ち合い、自分の陣地に落とされた方が敗者となります。

「ぺんぎんくんウォーズ」も撃ち合うゲームの一種だといえます。このゲームでは複数のボールを相手に押しつけ合います。すべてのボールを相手側に送り込めば勝ちです。ボールを相手に当てると、一定時間気絶させることもできます。

## アルゴリズム Algorithm

ものを撃ち合うミッションを実現するには、もの・ラケット・弾丸などの当たり判定処理がポイントになります。ラケットにものが当たった位置や、ものに弾丸が当たった位置によって跳ね返り方を変化させると、遊んでいて楽しい動きになるでしょう。

## まとめ Stage08

本章では、アクションゲームの核となる「ミッション」について解説しました。目的がなくてなんとなく遊ぶゲームも悪くないのですが、なんらかの目的があった方が、誰でも遊びやすいゲームになります。本章で紹介したミッションは単独でも使えますが、いくつかのミッションを組み合わせてもよいでしょう。世の中の多くのゲームでは、単純なミッションをいくつか組み合わせて、複雑なミッションを構成しています。

また、ゲームの操作方法を遊びながら覚えられるようにするには、ミッションの構成や難易度を工夫する必要があります。ゲームを始める前に、分厚い説明書を読まなければならないのはとても面倒です。ステージをクリアするごとにミッションがだんだん難しくなり、新しい操作やテクニックを少しずつ覚えられるようにすれば、プレイヤーはストレスなくゲームのルールを覚えていくことができます。

というわけで、「プレイヤーがゲームを遊びやすくなるようなミッションを作ろう！」というのが本章のまとめです。

・デモプログラム一覧
・引用ゲーム一覧
・索引

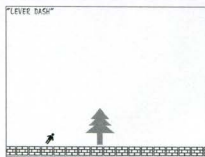
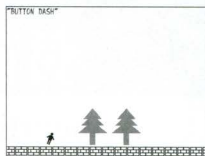
# 付録
## Appendix

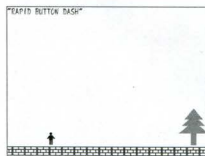
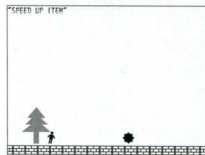
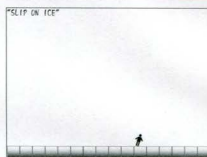


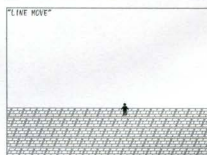
# Bonus Stage

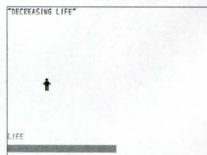
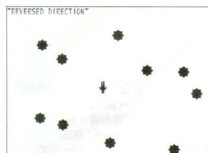
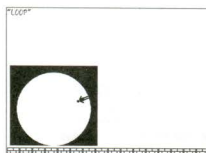
Appendix 1

# デモプログラム一覧

## Stage01　移動 Move

### LEVER DASH
→ p. 16
「レバーダッシュ」
→：移動

### BUTTON DASH
→ p. 20
「ボタンダッシュ」
→：移動
Z：加速

### DOUBLE LEVER DASH
→ p. 24
「レバー2段ダッシュ」
→　：移動
→→：加速

### RAPID BUTTON DASH
→ p. 27
「連打ダッシュ」
Z　：移動
Z連打：加速

### SPEED UP ITEM
→ p. 30
「スピードアップアイテム」
→：移動

### SLIP ON ICE
→ p. 33
「氷ですべる」
←→：移動

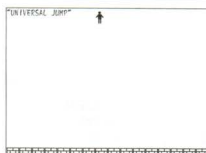
### SWIMMING
→ p. 37
「泳ぐ」
←→：移動
Z　：浮上

### LINE MOVE
→ p. 40
「ライン移動」
←→↑↓：移動

### SCREEN EDGE WARP
→ p. 43
「画面端ワープ」
←→↑↓：移動

### DECREASING LIFE
→ p. 47
「移動するとライフが減る」
←→↑↓：移動

**REVERSED DIRECTION**
→ p. 50
「入力と逆の方向へ動く」
←→↑↓：移動

**LOOP**
→ p. 53
「ループ」
→：移動

## Stage02 ジャンプ Jump

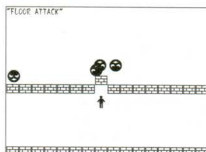
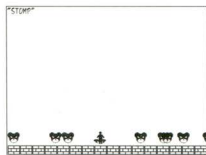
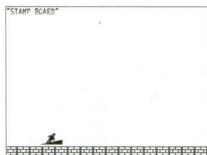
**FIXED JUMP**
→ p. 60
「固定長ジャンプ」
←→：移動
Z　：ジャンプ

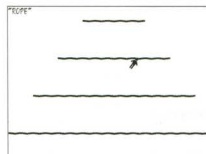
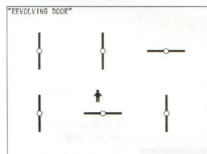
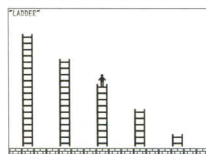
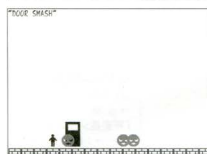
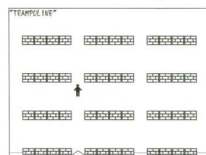
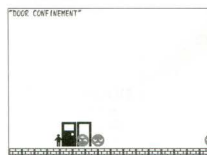
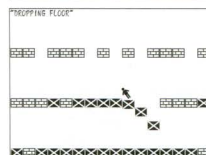
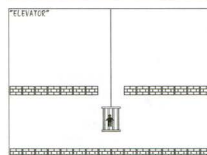
**JUMP OFF**
→ p. 83
「飛び降り」
←→：移動
Z　：ジャンプ
↓＋Z：飛び降り

**VARIABLE JUMP**
→ p. 66
「可変長ジャンプ」
←→：移動
Z　：ジャンプ

**UNIVERSAL JUMP**
→ p. 94
「ジャンプ飛行」
←→：移動
Z　：ジャンプ

**DOUBLE JUMP**
→ p. 72
「2段ジャンプ」
←→：移動
Z　：ジャンプ

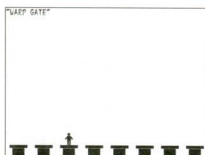
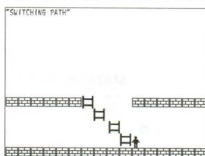
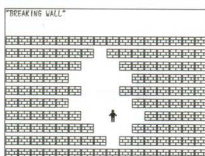
**JUMP ANGLE**
→ p. 99
「ジャンプ角度調整」
Z　：ジャンプ
Z連打：加速

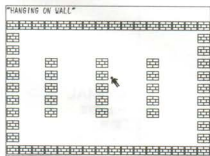
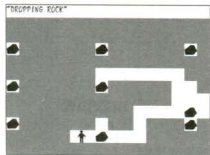
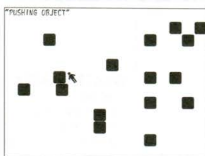
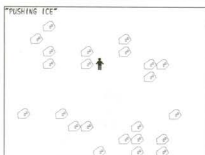
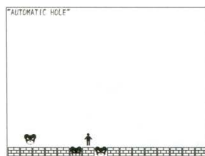
**TRIANGLE JUMP**
→ p. 77
「三角跳び」
←→：移動
Z　：ジャンプ

**FLOOR ATTACK**
→ p. 106
「床アタック」
←→：移動
Z　：ジャンプ

**STOMP**
→ p. 116
「踏みつけ」
←→：移動
Z　：ジャンプ

**STAMP BOARD**
→ p. 121
「踏み切り板」
←→：移動
Z　：ジャンプ

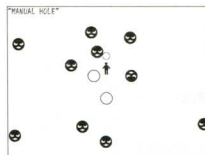
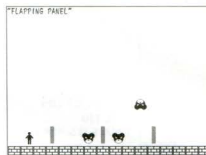
## Stage03　仕掛け Gimmick

**ROPE**
→ p. 124
「ロープ」
←→：移動
↓　：ロープを放す

**REVOLVING DOOR**
→ p. 143
「回転ドア」
←→↑↓：移動

**LADDER**
→ p. 128
「はしご」
←→↑↓：移動

**DOOR SMASH**
→ p. 148
「ドア飛ばし」
Z：ドアの開閉

**TRAMPOLINE**
→ p. 133
「トランポリン」
←→：移動

**DOOR CONFINEMENT**
→ p. 154
「ドア閉じ込め」
←→：移動

**DROPPING FLOOR**
→ p. 138
「抜ける床」
←→：移動

**ELEVATOR**
→ p. 160
「エレベーター」
←→：移動

**MOVING FLOOR**
→ p. 166
「動く足場」
←→：移動

**BELT CONVEYOR**
→ p. 170
「ベルトコンベア」
←→：移動

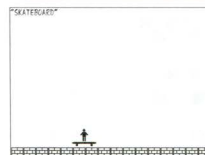
**AIR CURRENT**
→ p. 174
「上昇気流」
←→：移動

**WARP GATE**
→ p. 179
「ワープゲート」
←→：移動
↓ ：ゲートに入る

**SWITCHING PATH**
→ p. 183
「切り替わる通路」
←→↑↓：移動
Z ：ゲート切り替え

**BREAKING WALL**
→ p. 188
「壁を壊して通路を作る」
←→↑↓：移動
Z ：壁の破壊

## Stage04 地形利用 Land Features

**HANGING ON WALL**
→ p. 194
「壁や天井に張り付く」
←→↑↓：移動
Z ：ジャンプ
↓+Z ：飛び降り

**DROPPING ROCK**
→ p. 202
「岩落とし」
←→↑↓：移動

**PUSHING OBJECT**
→ p. 207
「ものを押して動かす」
←→↑↓：移動

**PUSHING ICE**
→ p. 211
「氷を押す」
←→↑↓：移動
Z ：氷を押す

AUTOMATIC HOLE
→ p. 216
「自動穴」
←→：移動
Z　：左へ穴を掘る
X　：右へ穴を掘る

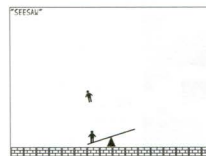
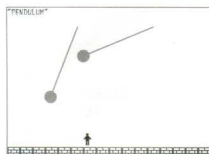
FOOTHOLD
→ p. 242
「足場を作る」
←→：移動
Z　：足場を作る

MANUAL HOLE
→ p. 226
「手動穴」
←→：移動
Z　：穴を掘る
X　：穴を埋める

FLAPPING PANEL
→ p. 247
「壁を倒す」
←→：移動
Z　：壁を起こす

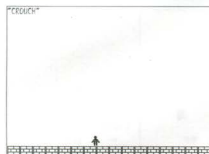
SETTING ROPE
→ p. 235
「ロープを張る」
←→↑↓：移動
Z　：ロープを張る

DROPPING LAND
→ p. 255
「地面を落とす」
←→↑↓：移動
Z　：ヒビを入れる

## Stage05 特殊行動 Special Motion

SKATEBOARD
→ p. 268
「スケートボード」
←→：移動
Z：スケートボードの乗降

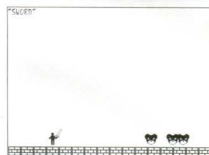
ANIMAL
→ p. 276
「動物」
←→：移動
Z　：動物の乗降

CAR
→ p. 271
「自動車」
←→：移動
Z　：ジャンプ

SEESAW
→ p. 280
「シーソー」
←→：移動

**PENDULUM**
→ p. 285
「振り子」
←→：移動
Z　：ジャンプ

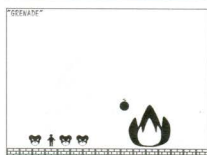
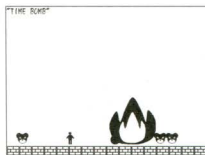
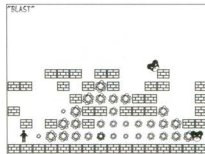
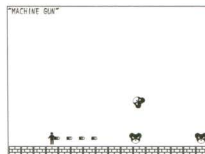
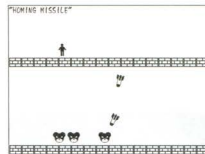
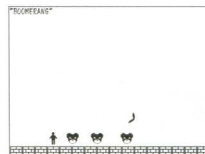
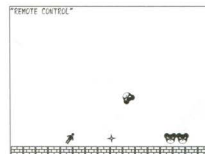
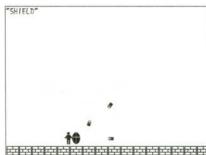
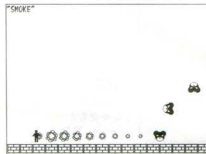
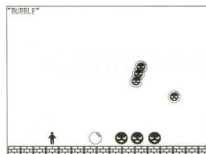
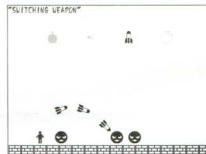
**GIANT**
→ p. 296
「巨大化」
←→：移動
Z　：巨大化

**CROUCH**
→ p. 291
「しゃがむ」
←→：移動
↓　：しゃがむ

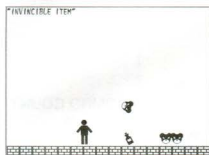
**MULTIPLE CHARACTER**
→ p. 299
「複数キャラクターの操作」
←→↑↓：移動
Z：キャラクター切り替え

**CURLING UP**
→ p. 294
「丸まる」
←→　：移動
↓　：しゃがむ
↓＋←→：丸まり移動

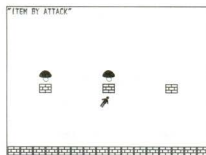
## Stage06 武器 Weapon

**SWORD**
→ p. 304
「剣」
←→：移動
Z　：攻撃

**BOUNCING BALL**
→ p. 314
「跳ねるボール」
←→：移動
Z　：攻撃

**WHIP**
→ p. 309
「ムチ」
←→：移動
Z　：攻撃

**GRENADE**
→ p. 317
「手榴弾」
←→：移動
Z　：攻撃

**TIME BOMB**
→ p. 321
「時限爆弾」
←→：移動
Z　：爆弾を設置

**BLAST**
→ p. 323
「爆煙」
←→：移動
Z　：爆弾を設置

**MACHINE GUN**
→ p. 329
「マシンガン」
←→：移動
Z　：攻撃

**HOMING MISSILE**
→ p. 332
「誘導ミサイル」
←→：移動
Z　：攻撃

**BOOMERANG**
→ p. 337
「ブーメラン」
←→：移動
Z　：攻撃

**REMOTE CONTROL**
→ p. 340
「リモコン武器」
←→　　：移動
Z　　　：攻撃
←→+Z：武器の操作

**SHIELD**
→ p. 343
「盾」
←→：移動
Z　：盾をかまえる

**SMOKE**
→ p. 345
「煙幕」
←→：移動
Z　：爆煙を出す

**BUBBLE**
→ p. 349
「泡」
←→：移動
Z　：攻撃
X　：ジャンプ

**SWITCHING WEAPON**
→ p. 355
「武器切り替え」
←→：移動
Z　：攻撃
X　：武器の選択

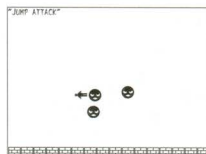
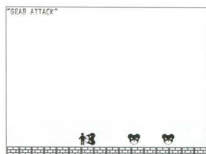
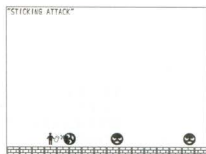
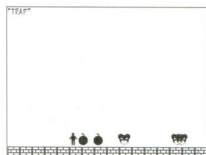
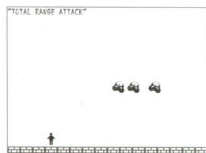
## Stage07　アイテム Item

INVINCIBLE ITEM
→ p. 362
「アイテムで無敵になる」
←→：移動

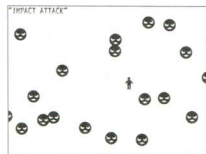
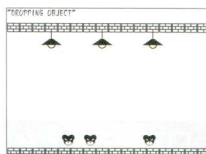
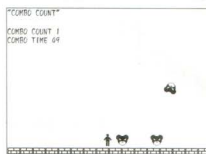
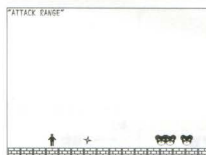
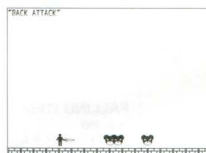
ITEM BY PASS
→ p. 372
「特定の場所を通るとアイテム出現」
←→：移動

THROWING
→ p. 365
「アイテムを拾って投げる」
←→：移動
Z　：アイテムを投げる

ITEM BY ATTACK
→ p. 375
「特定の場所を攻撃するとアイテム出現」
←→：移動
Z　：ジャンプ

FALLING ITEM
→ p. 369
「舞い落ちるアイテム」
←→：移動

## Extra Stage　攻撃 Attack

BODY ATTACK
→ CD-ROM
「体当たり攻撃」
←→：移動

JUMP ATTACK
→ CD-ROM
「ジャンプ攻撃」
←→：移動
Z　：ジャンプ
Z+X：ジャンプ攻撃

**GRAB ATTACK**
→ CD-ROM
「つかみ攻撃」
←→：移動
Z　：つかむ

**STICKING ATTACK**
→ CD-ROM
「足止め攻撃」
←→：移動
Z　：攻撃

**TRAP**
→ CD-ROM
「罠」
←→：移動
Z　：爆弾の設置

**TOTAL RANGE ATTACK**
→ CD-ROM
「全範囲攻撃」
←→：移動
Z　：攻撃

**CHARGE ATTACK**
→ CD-ROM
「ため攻撃」
←→：移動
Z　：攻撃

**IMPACT ATTACK**
→ CD-ROM
「突き飛ばし攻撃」
←→↑↓：移動

**DROPPING OBJECT**
→ CD-ROM
「落下物攻撃」
Z：落とす

**COMBO COUNT**
→ CD-ROM
「コンボ」
←→：移動

**ATTACK RANGE**
→ CD-ROM
「間合いで攻撃が変わる」
←→：移動
Z　：攻撃

**BACK ATTACK**
→ CD-ROM
「背後攻撃」
←→：移動
Z　：攻撃
Z+X：背後攻撃

**SHAKING LEVER**
→ CD-ROM
「レバガチャで敵を振りほどく」
←→：移動

Appendix 2

# 引用ゲーム一覧

本文中で引用したゲームの一覧表です。PCや家庭用ゲーム機に移植されている作品も多いので、気になるゲームはぜひ一度、実際に遊んでみることをお勧めします。なお、一部のゲームには続編が出ていたり、移植の際にタイトルが変わっていたりする場合があるので、詳しくは各メーカーのWebサイトやショップなどでお確かめください。

※凡例

■ゲーム名
・メーカー：発売年度：プラットフォーム
AC：アーケード、DC：ドリームキャスト、FC：ファミリーコンピュータ
GBA：ゲームボーイアドバンス、GC：ゲームキューブ、GW：ゲームウォッチ
MD：メガドライブ、NG：ネオジオ、PC：PC/AT互換機（Windowsパソコン）
PS：プレイステーション、PS2：プレイステーション2、PS3：プレイステーション3
PSP：プレイステーションポータブル、SFC：スーパーファミコン、SS：セガサターン
WII：Wii、XB：Xbox、XB360：Xbox360
・本書のなかで取り上げた主な特色
・コメント

## ■アイスクライマー

・任天堂：1985：AC、FC、GBA
・氷ですべる、画面端ワープ

ハンマーを使って氷の床を壊しながら、複数の階で構成された雪山を登っていくゲーム。すべる床や動く床の上をジャンプしながら登っていくので、精密な操作が要求されます。2人で同時に遊べるので、協力プレイや対戦プレイなど、工夫しだいでいろいろな遊び方ができます。

---

## ■悪魔城ドラキュラ（あくまじょうドラキュラ）

・コナミ：1986：AC、DS、FC、GBA、MD、N64、PS、PS2、SFC、SS、WII
・ムチ

ムチを片手にドラキュラの城に潜入し、さまざまな敵を倒しながら進んでいくゲーム。ムチはアイテムを取ることによって、攻撃力や射程距離がアップします。ムチは敵を攻撃するだけではなく、敵が投げてくる飛び道具などを落とすこともできます。

---

## ■アルゴスの戦士（アルゴスのせんし）

・テクモ：1986：AC、FC、PS2、XB
・ムチ（ヨーヨー）

ヨーヨーのような動きをする武器が印象的なゲーム。ヨーヨーはキャラクターの前方に伸ばすだけではなく、円形の軌道でキャラクターの頭上に振り回して、上からきた敵を攻撃することもできます。

■イーアルカンフー

・コナミ：1985：AC、DS、FC、GBA、PS

・三角跳び

カンフーを使う格闘家を操作して、さまざまな特徴を持った敵と戦うゲーム。敵によって動作や使う武器が異なるため、敵の種類に応じて攻略法を変える楽しみがあります。自分のキャラクターも、レバーとボタンの組み合わせによっていろいろな攻撃が出せます。最も初期の格闘ゲームの1つだといえます。

---

■イシターの復活（イシターのふっかつ）

・ナムコ：1986：AC、PS

・複数キャラクターの操作

「ドルアーガの塔」の続編に相当するゲーム。アーケードゲームながらもロールプレイングゲーム的な性格が強く、プレイ終了時に表示されたパスワードを次回プレイ時に入力することで、前回の続きからプレイを再開することができます。2人のキャラクターを、2つのレバーと2つのボタンで操作するのが特徴です。1人でプレイすることも、2人でプレイすることもできます。

---

■海腹川背（うみはらかわせ）

・TNN：1994：PS、SFC

・ロープを張る、ムチ（ロープ）

伸び縮みするロープを使って、非常に多彩なアクションが楽しめるゲーム。敵にロープを当てて攻撃するだけではなく、ものにひっかけてぶら下がったり、遠心力や張力を利用して高くジャンプしたりダッシュしたりと、かなり幅広いアクションが可能です。スムーズに伸び縮みするロープの動きは一見の価値があります。

---

■エレベーターアクション

・タイトー：1983：AC、GBA、GC、PS2、PSP

・エレベーター、落下物攻撃

ビルの屋上から潜入して、機密文書を入手し、ビルの最下階から脱出するゲーム。階の上下にはエレベーターを使います。敵もエレベーターを使って階を移動しながら、キャラクターを追いかけてきます。また、銃を使って天井の電灯を撃ち落とし、うまく頭上に落下させることによって、敵を倒すこともできます。

---

■オバケのQ太郎ワンワンパニック

・バンダイ：1985：FC

・移動するとライフが減る

藤子不二雄氏の漫画「オバケのQ太郎」を題材にしたゲーム。オバQを操作して、さらわれた仲間を助けるという設定です。見た目はかわいらしいゲームですが、かなり難易度は高めです。敵の攻撃が激しいだけではなく、オバQは移動するたびにどんどん空腹になっていくため、食べ物の回収にも気を配る必要があります。

---

■ガントレット

・アタリ：1985：AC、MD

・複数キャラクターの操作（複数人数プレイ）

最大4人まで同時に遊ぶことができるゲーム。ファンタジー世界をモチーフにしており、ウォリアー・ヴァルキ

リー・エルフといったキャラクターが登場します。次々と生成される膨大な数のモンスターを、複数のプレイヤーが協力して倒しながら、出口に向かうことが目的です。

## ■キューバート

**・Gottlieb：1982：AC、SFC**

**・ペイント**

キューブを積み重ねたような立体感のあるステージを跳ね回って、すべてのキューブを塗りつぶすゲーム。キューブを通ると塗りつぶすことができます。キャラクターや敵はキューブからキューブへジャンプして移動するので、うっかり敵と同じキューブにジャンプしないように、敵の動きを先読みしてプレイする必要があります。

## ■グラナド・エスパダ

**・imc GAMES/Hanbit：2003：PC**

**・複数キャラクターの操作**

3人のキャラクターを同時に操作することが特徴のMMORPGです。操作方法やキャラクターのAIがよく練られているため、アクションゲームのような激しい動きにもかかわらず、複数のキャラクターを同時に動かしていても十分にコントロールできます。グラフィックの品質がよく、膨大な数の敵が出現しても処理速度が落ちないなど、完成度の高いゲームです。

## ■クルクルランド

**・任天堂：1984：AC、FC、GBA**

**・特定の場所を通るとアイテム出現**

格子状にポールが並んだステージのなかを動き回って、隠された金塊をすべて見つけ出すゲーム。キャラクターが通過した場所に金塊があると、金塊を出現させることができます。金塊の配置は絵や模様になっているため、すべての金塊を出現させて、隠された絵を明らかにする楽しみがあります。キャラクターがポールにつかまって左右にターンする動きも特徴的です。

## ■クレイジーバルーン

**・タイトー：1980：AC**

**・目的地へいく**

風船を操作して、針のような壁が並んだ迷路を抜けるゲーム。風船が壁に当たると割れてしまいます。風船は振り子のように左右に揺れているため、狭い通路を通るときには、揺れのタイミングに合わせて上手に風船を移動させる必要があります。1995年頃にTV番組で放映され、後に玩具にもなった「電流イライラ棒」(玩具名は「電撃イライラ棒」)の原型のようなゲームです。

## ■ゲイングランド

**・セガ：1988：AC、MD**

**・目的地へいく、すべての敵を倒す**

20種類にも及ぶキャラクターを操作して、ステージ内の敵を倒しつつ、出口から脱出するゲーム。すべての敵を倒すか、すべてのキャラクターが出口から脱出すると、ステージクリアとなります。ステージごとに有効な攻め方が異なるため、戦術や用兵の研究が楽しめるゲームです。

## ■ゲゲゲの鬼太郎 妖怪大魔境（ゲゲゲのきたろう ようかいだいまきょう）

・バンダイ：1986：FC
・リモコン武器

水木しげる氏の漫画「ゲゲゲの鬼太郎」を題材にしたゲーム。原作に登場する「毛バリ」「ちゃんちゃんこ」「指鉄砲」「リモコン下駄」といった武器が使えるのが魅力です。また、ステージには妖界魔境や妖奇魔境といった複数の種類があり、ステージごとにクリアの条件が異なる点も楽しめます。

## ■源平討魔伝（げんぺいとうまでん）

・ナムコ：1986：AC、PS
・剣

死語の世界からよみがえった平景清が、源義経や弁慶と戦いながら、仇敵である源頼朝の討伐を目指すという設定のゲーム。平面のマップを歩くステージ、横スクロールのステージ、大きなキャラクターが登場するステージの3種類から構成されています。大きなキャラクターが剣で戦う様子は迫力があります。

## ■ゴールデンアックス

・セガ：1989：AC、MD、PS2、SS、WII
・レバー2段ダッシュ、ライン移動、動物、ジャンプ攻撃、つかみ攻撃（連続攻撃）、背後攻撃、剣、すべての敵を倒す

ファンタジー世界を題材にした、格闘要素が強いゲーム。横スクロールするステージを、剣や斧、魔法などを使って敵を倒しながら進んでいきます。レバーとボタンの組み合わせによって、ダッシュ攻撃やハイジャンプ攻撃など、多彩な攻撃が可能です。また、動物に乗っている敵をたたき落として、奪った動物に乗って逆に攻撃するといったこともできます。

## ■魂斗羅（こんとら）

・コナミ：1987：AC、DS、FC、GBA、MD、PS2、SFC、XBOX360
・飛び降り、マシンガン、武器切り替え

銃を抱えた兵士を操作して、さまざまな罠をくぐり抜けたり、敵を蹴散らしたりしながら進んでいくゲーム。武器の種類が多く、銃だけでも何種類もあるため、状況に応じて素早く武器を切り替えながら攻撃する必要があります。2人同時プレイでは、お互いに罠や敵の出現位置を覚えて、ぴったりと協力しながら進む必要があるため、緊張感のあるプレイが楽しめます。

## ■サーカス

・Exidy：1977：AC
・シーソー

シーソーを左右に動かして、落下してくる人をシーソーで跳ね返し、上空にある風船を取らせるゲーム。人はシーソーで跳ねるだけではなく、風船を取ったときにも反発するような動きをします。また、動きに応じて手足をばたつかせるため、見ているだけでなんとなくおかしさがこみ上げてくるコミカルなゲームです。

## ■サーカスチャーリー

**・コナミ：1984：AC、DS、FC、PS**

**・振り子**

さまざまなサーカスの演目をこなすゲーム。火の輪くぐり・綱渡り・空中ブランコなど6種類の演目があり、それぞれまったく違ったアクションが楽しめます。キャラクターや動物などのグラフィックはかわいらしいのですが、難易度が高いステージでは精密な操作が要求されます。特にジャンプのタイミングがシビアです。

---

## ■最後の忍道（さいごのにんどう）

**・アイレム：1988：AC**

**・ムチ（鎖鎌）、武器切り替え**

忍者が主人公のゲーム。刀・手裏剣・爆弾・鎖鎌といった4種類の武器を切り替えながら進みます。伸び縮みする鎖鎌は、レバー入力と組み合わせることによって発射方向を変えたり、円形に振り回したりすることができます。グラフィックが美しいゲームです。

---

## ■ザ・キャッスル

**・アスキー：1985：FC、PC**

**・ベルトコンベア**

魔王に連れ去られた姫を救うため、魔王の城に王子が単身で潜入するゲーム。パズル要素が強いステージが100面も用意されているので、謎解きはかなり歯ごたえがあります。後に、より難易度が高くなった「キャッスルエクセレント」も発売されました。パレット固定で8色しか出ない当時のPCのグラフィックにもかかわらず、巧みな色づかいで美しい画面を実現しています。

---

## ■サムライスピリッツ

**・SNK：1993：AC、MD、NG、PS、PS2、SFC**

**・三角跳び**

チャンバラを題材にした格闘ゲーム。パンチやキックを使って戦う多くの格闘ゲームとは違い、ほとんどのキャラクターが剣を使って戦います。剣での斬り合いのためか、攻撃一発あたりのダメージが大きく、タイミングによってはたった1、2発の攻撃で勝負が決まってしまうこともあります。一発の攻撃が大きいため、間合いを読み合うのがとても楽しいゲームです。

---

## ■シティコネクション

**・1985：ジャレコ：FC、PS**

**・ペイント**

クラリスという名の少女が運転するという設定の自動車を操作して、段差のあるステージを走り回り、すべての床を塗りつぶすゲーム。敵のパトカーや、ときどき出現する障害物の猫などは、方向転換やジャンプでかわします。ステージの背景には、マンハッタンや凱旋門など、世界中の風景が描かれています。

---

## ■忍（しのび）

・セガ：1987：AC、MD

・しゃがむ

忍者を操作して、敵を手裏剣で倒しながら、横スクロールするステージを進んでいくゲーム。物陰にしゃがんで敵の攻撃をかわしたり、屋根の上に跳び上がったりなど、忍者らしいアクションが楽しめます。メガドライブ版の「ザ・スーパー忍」は、アーケード版を発展させた作品です。ステージ数が多く、グラフィックやゲーム内容も完成度が高いため、メガドライブゲームのなかでも特に評価が高い作品の1つです。

## ■Shinobi（しのび）

・セガ：2002：PS2

・コンボ

「忍」「ザ・スーパー忍」「シャドウダンサー」などの「忍」シリーズを題材にした、3Dグラフィックスのゲーム。「殺陣」というルールが特徴で、一定の時間内に敵を連続して倒すと、攻撃力がアップします。上手に利用すると、ボスなどに大きなダメージを与えることができます。殺陣の演出や、壁を走り抜ける動きなど、見ているだけで格好いいゲームです。

## ■シャドウダンサー

・セガ：1989：AC、MD

・しゃがむ

「忍」の続編に相当するゲーム。手裏剣やしゃがみを駆使して敵を倒しながら進むのは前作と同じですが、この作品には忍犬が登場します。忍犬は敵にかみついて、敵の動きを一定時間止めてくれるため、忍犬を駆使した新たな攻略法が楽しめます。

## ■Joust（ジュースト）

・Williams：1982：AC、XB

・ジャンプ飛行

槍を持ち、ダチョウにまたがった騎士を操作して、敵と戦うゲームです。ステージには段差があり、キャラクターの動きには慣性が働くため、慎重な操作が要求されます。ボタンを連打している間だけジャンプ飛行ができるルールは、「バルーンファイト」に受け継がれています。

## ■獣王記（じゅうおうき）

・セガ：1988：AC、MD、PS2、WII

・巨大化（変身）

格闘要素の強い、横スクロールのゲーム。筋肉質なキャラクターを操作して、パンチやキックで敵を倒しながら進みます。パワーアップアイテムを取ると、より筋肉質になって強くなり、さらに取ると非常に強力な獣人に変身します。獣人の種類はステージによって異なり、狼や熊などがあります。変身の瞬間に獣の顔がアップで表示されるなど、どことなくコミカルなゲームです。

## ■新入社員とおる君（しんにゅうしゃいんとおるくん）

**・コナミ：1984：AC、FC**

**・すべてのアイテムを取る**

新入社員のとおる君を操作して、彼女とのデートに向かうために課長の追撃を逃れながら会社を抜け出すというゲーム。ステージ内のハートを集めるのがクリアの目的ですが、ただハートに触れればよいのではなく、ほかの社員をヒップアタックで椅子から突き落としたうえで回収する必要があるなど、過激な設定がおかしみを誘います。

## ■スーパーマリオブラザーズ

**・任天堂：1985：DS、FC、GBA、GC、SFC、WII**

**・レバーダッシュ、ボタンダッシュ、スピードアップアイテム、泳ぐ、可変長ジャンプ、踏みつけ、動く足場、ワープゲート、巨大化、特定の場所を攻撃するとアイテム出現**

ジャンプやダッシュを駆使して、さまざまな敵や罠を切り抜けながら、横スクロールするステージを進んでいくゲーム。世界中で大ヒットとなり、今でも続編や関連作品が数多く作られているほか、横スクロールアクションゲーム全般に大きな影響を与えた作品だといえます。昔の作品ながらも、非常に数多くの要素が盛り込まれています。

## ■ストライダー飛竜（ストライダーひりゅう）

**・カプコン：1989：AC、MD、PS**

**・壁や天井に張り付く**

暗殺者の主人公を操作して、起伏に富んだステージをクリアしていくゲーム。キャラクターは現代的な忍者といったところで、天井や壁にぶらさがったり、急斜面を駆け下りたりと、ダイナミックな動きが可能です。主な武器は剣で、敵や障害物を一刀両断する爽快感が楽しめます。

## ■スパイvsスパイ（スパイバーサススパイ）

**・ケムコ：1986：FC**

**・罠**

アメリカで人気があったコミックを題材にしたゲーム。FC版以前にも、アタリやセガマークⅢなどで発売されています。数々の部屋から構成されたステージを歩き回って、アイテムを探し出し、脱出することが目的です。部屋に爆弾・硫酸バケツ・毒ガスといった過激な罠を仕掛けて、ライバルを陥れることができます。2人対戦が楽しいゲームです。

## ■スパルタンX（スパルタンエックス）

**・アイレム：1984：AC、FC、PS、SS、WII**

**・ジャンプ攻撃、レバガチャで敵を振りほどく**

ジャッキーチェン主演の映画を題材にした、格闘要素の強いゲーム。敵のアジトに潜入し、ジャンプ・パンチ・キックを駆使して敵を倒しながら、ボスにさらわれたヒロインを助け出すことが目的です。敵が次々と出てくるため、油断するとすぐに敵にまとわりつかれてしまいますが、レバガチャで振りほどくことができます。

## ■ゼルダの伝説（ゼルダのでんせつ）

・任天堂：1986：DS、FC、GBA、GC、SFC、WII

・抜ける床、上昇気流、ブーメラン

アドベンチャー要素やパズル要素の強いアクションゲーム。主人公のリンクを操作して、数々の敵やアイテムが待ち受けるステージを冒険しながら、最終ボスを目指します。続編が多く作られており、主人公のリンクは共通ながらも、作品によってまったく違った冒険が楽しめます。GC版の「風のタクト」では、風を操れるタクト（指揮棒）を上手に使って、グライダーやヨットなどを操作するアクションが印象的です。

---

## ■ソニック・ザ・ヘッジホッグ

・セガ：1991：AC、DC、DS、GBA、GC、MD、PS2、PS3、SS、WII、XB、XB360

・レバーダッシュ、ループ、丸まる

ハリネズミのソニックを操作して、起伏に富んだステージを高速で駆け抜けるゲーム。スピード感のある展開が持ち味で、体を丸めることによって、坂やループといった仕掛けをハイスピードで通り抜けていくことができます。「スーパーマリオブラザーズ」を意識した作品ですが、動きや地形にはいろいろと新しい要素が盛り込まれています。

---

## ■ソンソン

・カプコン：1984：AC、FC、PS、PS2、PSP、SS

・飛び降り

西遊記を題材にした横スクロールシューティングゲーム。複数の階で構成されたステージを上下に移動しながら、迫ってくる敵をショットで倒して進みます。レバー操作で階を上下する動きは、ほかのシューティングゲームにはあまり例を見ません。主人公や敵のグラフィックがかわいらしいほか、アイテムの食べ物の種類がとても多く、遊んでいるとおなかがすくゲームです。

---

## ■ダイナマイトダックス

・セガ：1989：AC

・ため攻撃

あひるを主人公にした、格闘要素の強いアクションゲーム。横スクロールするステージを、パンチやライン移動を駆使して、敵を倒しながら進んでいきます。攻撃ボタンを押しっぱなしにすることでパワーをため、強力なため攻撃を放つことができます。腕を回してパワーをためる様子がコミカルです。

---

## ■ダイナマイト刑事（ダイナマイトでか）

・セガ：1996：DC、SS、PS2

・武器切り替え、アイテムを拾って投げる

過激な刑事が主人公のゲーム。ステージに配置されたありとあらゆるものを拾って武器にできることが特徴です。銃やナイフといった普通の武器だけではなく、モップや柱時計、はてはコショウ・桃・寿司船・冷凍マグロといった本来は武器ではないものまでも、攻撃に使うことができます。例えば寿司船などは、まず寿司を1個ずつ投げてから、最後に空になった船を投げつけるなど、芸が細かいゲームです。

---

## ■戦いの挽歌（たたかいのばんか）

・コナミ：1986：AC
・盾

剣と盾を使って敵を倒しながら進んでいく、横スクロールのアクションゲーム。ボタンで盾をかまえることによって、敵の打撃や飛び道具を防ぐことができます。盾で防御し、敵の攻撃の隙をついて剣を振るという切り替えが楽しいゲームです。

---

## ■タッパー

・Bally Midway：1983：AC
・待ち行列の処理

酒場を題材にしたゲーム。バーテンダーを操作して、次々と訪れる客に対してビールをサービスします。カウンター上をジョッキをすべらせて客に渡すと、客は飲み終えたジョッキをすべらせて返してくるので、落ちないように受け止める必要があります。どの客に先にサービスするのか、順番をよく考えて素早く反応することがゲームのポイントです。

---

## ■チェルノブ

・データイースト：1988：AC、MD
・誘導ミサイル、ブーメラン、武器切り替え

地上を走ったりジャンプしたりする人間が自機のゲーム。強制横スクロールに左右移動が組み合わさっていたり、ジャンプが2種類あったり、ショットの方向が変えられたりと、操作性は独特です。「戦う人間発電所」というやや危険な副題がついていたり、円高やドル安といった名前のついたアイテムが出たりと、当時の世界情勢が反映されています。

---

## ■ちゃっくんぽっぷ

・タイトー：1983：AC、FC、PC、PS2、PSP
・壁を壊して通路を作る、壁や天井に貼り付く、跳ねるボール（爆弾）、爆煙、アイテムで無敵になる

主人公の「ちゃっくん」を操作して、迷路のなかに捕らわれたハートを救出し、出口から脱出するゲーム。爆弾を使って敵の「もんすた」を倒したり、壁を壊したりすることができます。爆弾を出すと少し転がってから爆発し、爆煙が広がります。複数の爆弾を狭い地形で爆発させると爆煙が大きく広がるなど、爆弾の使い方にはいろいろとテクニックがあります。

---

## ■ディグダグ

・ナムコ：1982：AC、DS、FC、PS、PS2、PSP
・壁を壊して通路を作る、岩落とし

ドリルを持った主人公を操作して地中を掘り進み、モンスターを倒すゲーム。地面を掘ることで、自由な形に通路を作れます。これを利用して敵を岩の下に誘い込んだり、薄い壁越しに敵を攻撃したりと、いろいろなテクニックが楽しめます。

---

## ■ディグダグⅡ（ディグダグツー）

・ナムコ：1985：AC、DS、PSP

・地面を落とす

「ディグダグ」の続編。本作では小さな島の地上が舞台になっています。島にひびを入れると、島の一部を切り崩して海中に落とすことができます。敵をポンプでふくらませて動けなくさせておき、その間に島を切り崩して敵ごと海に落とすといった、前作同様のテクニカルなアクションが楽しめます。

---

## ■10ヤードファイト（テンヤードファイト）

・アイレム：1983：AC

・レバガチャで敵を振りほどく

アメリカンフットボールを題材にしたゲーム。「ロードランナー」などで有名な米国ブローダーバンド社の作品を、アイレムがアーケードゲーム化したものです。敵をかわしたり、まとわりついてくる敵をレバガチャで振りほどいたりしながら、敵の陣地に攻め込みます。

---

## ■ドアドア

・エニックス：1983：FC、PC

・ドア閉じ込め

チュンソフトの中村光一氏の出世作。主人公のチュン君を操作し、迫ってくるモンスターを、引き戸状のドアを使って退治します。ドアを開けておき、敵を誘い込んでから閉めると、敵を閉じ込めて倒すことができます。取っ手の位置によってドアの開く方向が違うので、ステージによって攻略法をよく考える必要があります。

---

## ■Dog Patch（ドッグパッチ）

・Midway：1977：AC

・ものを撃ち合う

中央に投げ上げられた缶を、左右から銃で撃ち合って、相手の陣地に落とすゲーム。弾の当たり方によって缶が左右に動いたり上に跳ね上げられたりと、なかなか思ったように動かない点がスリリングです。

---

## ■ドラゴンバスター

・ナムコ：1985：AC、FC、PS、PSP

・2段ジャンプ

ファンタジー世界を題材にしたゲーム。ドラゴンに人質にされた王女を助けるため、主人公が剣を片手に洞窟や城などに潜入します。ジャンプやしゃがみなどの動きと、剣や魔法による攻撃を駆使して、数々の敵を倒しながら進みます。精密なジャンプ操作と、間合いの見切りが要求されるゲームです。

---

## ■トラップガンナー

・アトラス：1998：PS

・罠

罠を仕掛けてライバルを倒すことが目的の、対戦要素の強いアクションゲーム。内容としては「スパイvsスパイ」に似ていますが、斜め上から見下ろした画面構成になっているため、ゲーム性はかなり違います。罠を仕掛けるだけではなく、自分に有利な陣地を作ることが攻略のポイントです。

---

## ■ドルアーガの塔（ドルアーガのとう）

・ナムコ：1984：AC、FC、PS
・剣

騎士を操作して、60階にも及ぶステージを進み、塔の最上階にいる巫女を助け出すゲーム。ステージの多さもさることながら、ステージごとに隠されたアイテムを回収しながら進む必要があるので、難易度は高めです。アイテムの出現方法には、特定の敵を指定の数だけ倒すといったものから、スタートボタンを押すといったとんでもない条件もあり、攻略するには出現方法を記憶しておく必要があります。

## ■ドンキーコング

・任天堂：1981：AC、DS、FC、GBA、GW、SFC、WII
・固定長ジャンプ、はしご

複数の階からなるステージを、転がってくるタルを避けながら登っていくゲーム。最上階にいるドンキーコングにさらわれた恋人を助けるのがゲームの目的です。タルを避けるには、タルの直前でタイミングよくジャンプする方法と、はしごを使ってタルをやりすごす方法があります。また、ハンマーを取ると、タルをたたいて壊すことができます。

## ■ドンキーコングJr.（ドンキーコングジュニア）

・任天堂：1982：AC、FC
・ロープ

「ドンキーコング」の関連作品。本作品の主人公は、ドンキーコングの息子という設定のドンキーコングJr.です。ステージに配置されたロープを登ったり、ロープからロープに飛び移ったりしながら、捕らわれの身になったドンキーコングを助けにいきます。ロープに片手でつかまったときと、両手でつかまったときに、移動速度が変わるのが面白いところです。関連作品に「ドンキーコングJr.の算数遊び」もあります。

## ■謎の村雨城（なぞのむらさめじょう）

・任天堂：1986：FC、GBA
・間合いで攻撃が変わる

ファミコンディスクシステム初期のゲーム。主人公の剣士を操作して、村雨城と呼ばれる城に潜入し、敵を倒したりかわしたりしながら城主を倒すことが目的です。主人公の武器は刀ですが、敵との距離に応じて、直接斬る攻撃と刀を投げる攻撃が切り替わります。激しい敵の攻撃をかわしつつ、敵に接近して攻撃する必要があるため、難易度は高めです。

## ■ナッツ&ミルク（ナッツアンドミルク）

・ハドソン：1984：FC
・トランポリン

固定画面で、パズル要素の強いアクションゲーム。主人公のミルクを操作して、敵のナッツを避けながらフルーツを回収し、恋人ヨーグルの家に向かうのが目的です。ゲームの内容としては「ロードランナー」に似ていますが、本作品はジャンプが中心なので、ゲーム性はまったく違います。かわいいキャラクターやトランポリンなどの仕掛け、バリエーション豊かなステージ構成が魅力です。ステージのエディット機能もあります。

## ■ニンジャウォリアーズ

・タイトー：1987：AC、MD

・しゃがむ

「ダライアス」の3画面筐体を使ったゲーム。主人公の忍者を操作して、手裏剣を武器に敵を倒しつつ進んでいきます。内容としては一般的な格闘要素の強い横スクロールアクションゲームですが、3枚の画面が横に並んでいるため、ほかのゲームにはない迫力と前方の見通しのよさがあります。

---

## ■忍者くん（にんじゃくん）

・UPL：1984：AC、FC

・飛び降り、体当たり攻撃

主人公の忍者くんを操作して、複数の階から構成されたステージを敵を倒しつつ進んでいくゲーム。正式名称は「忍者くん 魔城の冒険」ですが、「忍者くん」といえばこの作品を指します。上の階に跳び上がるときや下の階に飛び降りるときに敵に体当たりすると、一定時間気絶させることができます。武器は手裏剣ですが、上手に戦うには体当たりの併用がポイントになります。

---

## ■忍者龍剣伝（にんじゃりゅうけんでん）

・テクモ：1988：AC、FC

・壁や天井に張り付く

アメリカをモチーフにしたステージを舞台に、主人公の忍者を操作して、敵を倒しながら進んでいくゲーム。壁や天井に向かってジャンプし、特定のボタンを押すと、つかまることができます。壁につかまったままジャンプして、さらに高い場所に登るなど、忍者らしいアクションが楽しめます。関連作品にXbox用の「NINJA GAIDEN」があります。

---

## ■ハイパーオリンピック

・コナミ：1983：DS、FC

・連打ダッシュ、ジャンプ角度調整

オリンピックのさまざまな競技を題材にしたゲーム。走るボタン2つとジャンプボタン1つの、3つのボタンを使います。100メートル走・走り幅跳び・走り高跳びなど、6種類の競技があります。多くの競技は、走るボタンを連打して選手を走らせ、タイミングよくジャンプボタンを押して、跳んだりものを投げたりする方式になっています。ボタンを連打すること自体をゲームにした代表的なゲームです。

---

## ■バーガータイム

・データイースト：1982：AC、FC、PS2

・岩落とし、足止め攻撃

主人公のコックを移動させ、ステージに配置された4つのハンバーガーを完成させるゲーム。パンや肉といったハンバーガーの具材はステージの上下に分散して配置されていて、コックが上を通ると落下します。このとき下に敵がいると、具材に敵をはさんで倒すことができます。敵も目玉焼きやウインナーといった食べ物の形をしているので、遊んでいるとハンバーガーが食べたくなるゲームです。

---

## ■バーニンラバー

- **データイースト：1982：**AC、FC
- **レバーダッシュ**

高速で縦スクロールするステージのなかを、車を操作して進んでいくゲーム。いっしょに走行している敵の車にぶつかっても大丈夫ですが、自分の車が跳ね返されるため、壁に当たったり海に落ちたりしないように注意が必要です。レバー上下で車の速度を調整し、ボタンでジャンプするという単純な操作ですが、非常にスピード感があるので楽しめます。

---

## ■パックマン

- **ナムコ：1980：**AC、FC、PS、PSP
- **画面端ワープ、体当たり攻撃、アイテムで無敵になる、すべてのアイテムを取る**

主人公のパックマンを操作して、迷路のなかに配置されたエサを食べるゲーム。エサをすべて食べるとステージクリアです。襲ってくるモンスターに当たるとミスになりますが、パワーエサを食べると、一定時間は逆にモンスターを食べて倒すことができます。登場する4色のモンスターは、色によって性格が違います。ドットイーターゲームの代表格といえる作品です。

---

## ■パックランド

- **ナムコ：1984：**AC、FC、PC、PS
- **連打ダッシュ、ジャンプ飛行、踏み切り板、ものを押して動かす、自動車、特定の場所を通るとアイテム出現**

「パックマン」に登場するパックマンが主人公の横スクロールアクションゲーム。街・森・渓谷といった変化に富んだステージを、ダッシュやジャンプを駆使して走り抜け、妖精の国を目指します。往路の3ステージと、帰路の1ステージが1セットになっており、帰路ではジャンプ飛行が可能です。左移動・右移動・ジャンプの3つのボタンを使う、独特の操作系を採用しています。

---

## ■バブルボブル

- **タイトー：1986：**AC、DS、GBA、PC
- **足場を作る、泡**

かわいらしいキャラクターが登場する固定画面のアクションゲーム。主人公はドラゴンのバブルンとボブルンで、泡を吐いて敵を倒したり、泡を足場にしてジャンプしたりすることができます。泡を敵に当てると、敵が泡に包まれてゆっくりと上昇します。敵が出てこないうちに素早くジャンプして泡を割ることで、敵を完全に倒すことができます。

---

## ■バルーンファイト

- **任天堂：1984：**AC、DS、FC、GBA
- **ジャンプ飛行**

体に風船をつけた主人公を操作して、やはり体に風船をつけた敵との空中戦を繰り広げるゲーム。「Joust」をアレンジした作品だといわれています。ボタンを連打することで主人公は上昇し、連打をやめるとゆっくりと下降します。2人同時プレイが可能で、協力プレイも対戦プレイも楽しめます。協力プレイの際には、うっかり仲間の風船を割らないように注意が必要です。

---

## ■ピットフォール2

・セガ：1985：AC、FC

・振り子

主人公の冒険家を操作して、さまざまなトラップをくぐり抜け、財宝を探し出すゲーム。アクティビジョン社によるAppleII向けのゲームをアーケードゲーム化したものです。画面は固定方式で、画面端にたどりつくと次の画面に進みます。ロープやトロッコなど、いかにも冒険らしい仕掛けがいろいろと用意されています。

## ■ファイナルファイト

・カプコン：1989：AC、GBA、MD、PS2、SFC

・ライン移動、三角跳び、ジャンプ攻撃、つかみ攻撃

悪の組織にさらわれたヒロインを倒すため、パンチやキック、ときには拾ったナイフや鉄パイプといった武器を駆使しながら、群がる敵を倒しつつ進んでいくゲーム。格闘系の横スクロールアクションゲームのなかでも、最も有名な作品の1つです。2人同時プレイも可能で、次々に襲いかかる敵を2人で分担して倒していく協力プレイには、1人プレイでは得られない楽しさがあります。

## ■フェアリーランドストーリー

・タイトー：1985：AC、PS2、PSP

・ものを押して動かす

かわいらしいキャラクターが登場する、固定画面のアクションゲーム。魔法使いの少女を操作して、ステージ内の敵をすべて倒すことが目的です。敵に魔法を当てるとケーキに変化させることができ、そのケーキを押して床の端から落とすと、敵を倒すことができます。ケーキを別の敵の上に落とすと、その敵も倒すことが可能です。

## ■ぶたさん

・ジャレコ：1987：AC

・アイテムを拾って投げる

主人公の豚を操作して、ステージに落ちている爆弾を拾って投げ、ライバルの豚を倒すゲーム。投げた爆弾を直接当てるか、時間で爆発したときに巻き込むと、ライバルを倒すことができます。爆弾を飛ばす距離は、投げるボタンを押す長さによって調整することが可能です。また、ボタンで伏せることができ、飛んできた爆弾をかわすことができます。

## ■フラッピー

・デービーソフト：1983：FC、PC

・ものを押して動かす、足止め攻撃

ヒヨコのような容姿の主人公フラッピーを動かして、ブルーストーンと呼ばれる青い石を、ブルーエリアと呼ばれる青い床の上まで運ぶことが目的のゲーム。ステージの数が多く、敵をかわすアクションが中心の面と、ブルーストーンを運ぶパズルが中心の面があり、攻略のしがいがあるゲームです。フラッピーに加えて、エビーラやユニコーンといった敵のかわいらしさも魅力です。

## ■フリスキートム

・日本物産：1981：AC、PS、SFC
・目的地へいく

ステージに配置された水道管を修理するゲーム。ネズミが水道管をかじると部品が床に落ちるので、落ちた部品を拾って、壊れた箇所まで水道管を伝って移動し修理します。水道管を修理して、多くの水を流すことが目的です。水道管の先には風呂が設置されていて、水を十分にためると美女(といっても当時の素朴なグラフィックですが)の入浴シーンが表示されるという、ちょっとしたごほうびが用意されています。

## ■フリッキー

・セガ：1984：AC、MD、PC
・味方を助ける

親鳥を操作して、さらわれたヒヨコを助けるゲーム。ヒヨコはステージ内の各所に配置されていて、親鳥が触れると後をついてきます。たくさんのヒヨコを一度に助けると高得点が得られますが、ヒヨコが敵の猫に触れると離れてしまうため、多くのヒヨコを無事に出口まで連れていくのは大変です。ヒヨコをたくさん連れていると、親鳥の後にヒヨコが美しい軌跡を描きます。

## ■プリンス・オブ・ペルシャ

・ブローダーバンド：1989：FC、MD、PC、SFC
・抜ける床

異国情緒のある美しいグラフィックと、滑らかなアニメーションが魅力のゲーム。大臣に陥れられた主人公の王子が、姫を助け出すために数々の罠を切り抜け、王宮の最上階を目指します。王宮のなかには、踏むと抜ける床や、床から飛び出てくる針、ギロチンなど、見た目からして危険な罠が張り巡らされています。また、深い谷間をダッシュとジャンプで跳び越えて、対岸にぶらさがってよじ登る、といったジャンプのアクションも豊富です。

## ■フロッガー

・コナミ：1981：AC、GBA、PS2
・目的地へいく

固定画面のアクションゲーム。カエルを操作して、車が走る道路を渡り、川を流れる丸太や亀の背中を飛び移って、無事に巣までたどりつくのが目的です。レバーで上下左右に移動するだけのシンプルな操作系で、ルールも単純ですが、ついつい遊び込んでしまう不思議な魅力があります。

## ■フロントライン

・タイトー：1983：AC、PS2
・動物(戦車)

兵士を操作して、敵兵や敵戦車が待ち受けるステージをくぐり抜け、敵の司令部を目指すゲーム。基本の武器は銃と手榴弾ですが、ステージに停まっている戦車や装甲車に乗り込み、武器として使うこともできます。戦車や装甲車が被弾したときには煙が出てしまいますが、一度降りてまた乗り直すことによって、煙を止めるテクニックもあります。

## ■平安京エイリアン（へいあんきょうエイリアン）

・電気音響：1979：AC、PC

・手動穴、すべての敵を倒す

東京大学の理論科学グループ（TSG）というサークルが開発し、話題を呼んだゲーム。平安京をモチーフにした格子状のステージを舞台に、主人公の検非違使（けびいし）が、穴掘りを武器にエイリアンを退治します。穴を掘って、エイリアンが落ちたときに素早く埋めることによってエイリアンを倒すことができます。戦術に「秋葉掘り」や「隠居掘り」といったいろいろな名前がついているのも面白い点です。2人同時プレイも可能です。

## ■ぺったんピュー

・サン電子：1984：AC

・壁を倒す

シルクハットをかぶったロボットのぺったんを操作して、ステージに配置されたパネルを使って敵を倒すゲーム。一定数の敵を倒すとステージクリアとなります。パネルを押すと倒すことができ、下にいる敵をつぶして倒すことができます。また、ボタンを押すと倒れたパネルがいっせいに起き上がり、上に敵が乗っていれば跳ね飛ばして倒すことができます。画面は疑似3Dで立体感があります。

## ■ぺんぎんくんウォーズ

・UPL：1985：AC、FC

・アイテムを拾って投げる、ものを撃ち合う

主人公のペンギンを操作して、ライバルの動物たちとボールを相手側に送り込む競技を行うゲーム。卓球のようなステージに10個のボールが配置されており、ボールの近くでボタンを押すと、相手側に送り込むことができます。すべてのボールを相手側に送り込んだ側の勝ちです。ボールを当てると相手は気絶して一定時間動けなくなります。気絶を使いこなすのが攻略のポイントです。

## ■ペンゴ

・セガ：1982：AC、MD、PC

・氷を押す

主人公のペンギンを操作して、ステージ内の敵を倒すゲーム。ステージには氷が迷路状に配置されており、接触してボタンを押すと氷をすべらせることができます。すべった氷で敵を押しつぶすと倒すことができます。また、ダイヤモンドが描かれた特殊なブロックがステージに3個あり、これらのブロックを一列に並べるとボーナスが入ります。

## ■ポップフレイマー

・ジャレコ：1982：AC

・ムチ（火炎放射器）

火炎放射器を持った主人公のネズミを操作して、ステージに配置された風船を割るゲーム。迫ってくる敵は、ムチのように炎が伸びる火炎放射器を使って倒すことができます。火炎放射器は使うたびに炎の長さが短くなっていきますが、風船を割ると回復します。また、ステージに配置されたジュースを飲むとパワーアップして、一定時間無敵になります。ジュースは一瞬で食べられる「パックマン」のパワーエサとは違い、飲むのに時間がかかります。

## ■ポパイ

**・任天堂：1982：AC、FC**

**・シーソー、舞い落ちるアイテム**

漫画やアニメでおなじみの「ポパイ」を題材にしたゲーム。ポパイを操作して、ブルートの攻撃をかわしながら、オリーブが投げるハートや音符などのアイテムを受け止めます。ひらひらと舞いながら落下するアイテムは、ステージの上の方で受け止めるほど高得点です。漫画やアニメのとおり、ホウレンソウを食べると一定時間パワーアップして、ブルートに体当たりして倒すことができます。

## ■Pong（ポン）

**・アタリ：1972：AC**

**・ものを撃ち合う**

卓球を題材にしたゲーム。史上初のビデオゲームとして知られているゲームです。ラケットを上下に操作して、斜めに飛んでくるボールを打ち返します。ボールを打ち返しそこなうと負けです。単純なゲームですが、ラケットにボールを当てた位置によって、ボールの角度や速度が大きく変わることが、このゲームを面白くしています。

## ■ボンバーマン

**・ハドソン：1985：DS、FC、PS、PS2、SFC、SS、XB360**

**・入力と逆の方向へ動く、壁を壊して通路を作る、時限爆弾**

迷路状のステージを歩き回り、爆弾を使ってライバルを倒すゲーム。ボタンを押すと置くことができる爆弾は、一定時間が経過すると自動的に爆発し、ライバルを爆風に巻き込むと倒すことができます。アイテムを取ることによって爆風の範囲が広がったり、移動速度が速くなったりします。爆風に巻き込まれた爆弾は誘爆するので、うっかり自分の爆風に巻き込まれないように、タイミングよく通路のかげに隠れるのが基本テクニックです。対戦プレイが熱いゲームです。

## ■マーブルマッドネス

**・アタリ：1984：AC、MD**

**・目的地へいく**

疑似3Dで描かれた立体感のあるステージのなかを、ビー玉を転がしてゴールまで導くゲーム。アーケード版はトラックボールを使ってビー玉を操作するため、ジョイスティックとは異なる繊細な操作を楽しむことができます。ステージには起伏や落とし穴、急な坂や細い橋などがあり、さらに敵も出現するので、操作する手にもついつい力が入ります。

## ■魔界村（まかいむら）

**・カプコン：1985：AC、FC、GBA、PS、PS2、PSP、SS**

**・手榴弾（たいまつ）、武器切り替え**

主人公の騎士を操作して、数々の敵を倒しながら、姫を助け出すために魔王の城を目指すゲーム。標準の武器は槍ですが、アイテムを拾うことによって、短剣・たいまつ・斧といった別の武器に切り替えることができます。武器によって性質がまったく違うため、状況に適した武器を拾う必要があります。主人公は鎧を着ており、敵の攻撃に当たるとパンツ一丁になってしまいますが、再び鎧を拾えば復活することができます。

## ■大魔界村（だいまかいむら）

**・カプコン：1988：AC、MD、PS、PS2、PSP、SS**

**・手榴弾（ナパーム）、武器切り替え**

「魔界村」の続編。前作で姫を助け出し、主人公は安息の日々を送っていたのですが、姫は別の魔王に魂を奪われてしまいます。姫の魂を奪い返すため、主人公は再び戦いに臨みます。基本的なルールは前作と同じですが、武器の種類が増えたり、巨大な敵が登場したりと、より派手な作品に進化しています。また、攻撃ボタンを押しっぱなしにしてパワーを溜めることによって、武器ごとに異なる魔法を使うことができます。

---

## ■マッピー

**・ナムコ：1983：AC、FC、GBA、PS**

**・トランポリン、抜ける床、ドア飛ばし、すべてのアイテムを取る**

ネズミの警官を操作して、ネコの窃盗団をかわしながら、盗品を取り返すゲーム。ステージは複数の階から構成されていて、階の移動にはトランポリンを使います。通路で敵に接触するとミスになりますが、トランポリンで上下に移動している間は接触しても大丈夫なので、トランポリンを上手に使って敵をやりすごすのが基本です。また、ドアを使って敵を跳ね飛ばしたり、ドアで自分を跳ね飛ばしてダッシュしたりと、いろいろなテクニックがあります。

---

## ■マリオブラザーズ

**・任天堂：1983：AC、FC、GBA、SFC、WII**

**・床アタック、全範囲攻撃**

「スーパーマリオブラザーズ」に先行する作品。マリオやルイージといったキャラクターが登場します。床を突き上げて敵を転ばせたあとに、敵に体当たりすると敵を蹴飛ばして倒すことができます。敵にはカメ・カニ・ハエが登場しますが、カニは2回突き上げないと転ばないとか、ハエは床近くにいるときしか転ばないといった、性格の違いがあります。また、「POW」と書かれたブロックを突き上げると、画面内のすべての敵を転ばせることができます。2人同時プレイも可能です。

---

## ■Mr.Do!（ミスタードゥ）

**・ユニバーサル：1982：AC、NG、SFC**

**・切り替わる通路、跳ねるボール**

ピエロのMr.Do!が主人公のゲーム。シリーズ作品が数多くあります。既存のゲームをベースに独自の要素を追加した作品が多く、例えば「ディグダグ」に似た「Mr.Do!」や、「ロードランナー」に似た「Mr.Do!キャッスル」などがあります。「Mr.Do!」では、「ディグダグ」と同じように穴を掘って通路を作ったり、りんごを敵の上に落として倒したりしますが、このほかに通路のなかを跳ね回るボールで攻撃することもできます。

---

## ■ミッキーマウス

**・任天堂：1981：GW**

**・待ち行列の処理**

ファミコン以前にブームとなったゲームウォッチのなかの一作品。ディズニーのミッキーマウスを題材にしています。主人公のミッキーマウスを操作して、4箇所から転げ落ちてくる卵をバスケットで受け止めます。卵が落ちてくる順番を覚えておき、順番にしたがって素早く受け止めないと、卵は床に落ちて割れてしまいます。スコアが上がるごとに卵が速くなり、後半はほとんど卵が見えないほどの速度になります。

---

## ■メタルスラッグ

**・SNK：1996：**AC、GBA、NG、PS、PS2、PSP、SS、XB

**・動物（戦車）、手榴弾**

緻密に描かれたグラフィックやアニメーションが魅力の、横スクロールアクションゲーム。多くのシリーズ作品が発売されています。主人公の兵士を操作して、ハンドガン・ナイフ・手榴弾といった武器を駆使しながら、襲いかかる無数の敵を倒しつつ進みます。武器にはこのほかにも、アイテムを取ることによって使えるマシンガンやロケットランチャー、乗り込むことによって使える戦車や潜水艦など、さまざまな種類があります。2人同時プレイも可能です。

---

## ■メトロイド

**・任天堂：1986：**DS、FC、GBA、GC、SFC

**・丸まる**

武器やアクションを駆使しつつ、さまざまな謎を解きながら、ステージを攻略していくゲーム。主な武器はビームですが、特定のアイテムを取ると回転ジャンプで体当たり攻撃をすることもできます。設定には映画「エイリアン」の影響が見られ、ドロドロとした暗い色調のグラフィックが、独特の雰囲気を醸し出しています。2Dアクションの「メトロイド」シリーズのほかに、3DのFPS（一人称視点シューティングゲーム）の「メトロイドプライム」シリーズがあります。

---

## ■メトロクロス

**・ナムコ：1985：**AC、FC、PS

**・ライン移動、踏み切り板、スケートボード**

主人公のランナーを操作して、近未来をモチーフにしたステージを制限時間内に駆け抜けるゲーム。ステージには、ジャンプして跳び越える必要があるハードルや、通ると破裂するクラッカーなど、さまざまな罠が仕掛けられています。また、踏むと制限時間の減少が止まる空き缶や、乗るとスピードアップするスケートボードなど、主人公を助けるアイテムもいろいろとあります。

---

## ■モトス

**・ナムコ：1985：**AC、PC、PS、PSP

**・突き飛ばし攻撃**

固定画面のステージ内で自機を操作し、敵に体当たりして床から落とすゲーム。床の端や穴の近くで上手に敵を突き飛ばすと、敵を落とすことができます。ただし、敵に接触すると自機も跳ね返されるので、逆に落とされてしまうこともあります。敵は自機を床から落とそうと、有利な位置に回り込みながら襲いかかってきます。アイテムを取ると、自機の重量を増やして落とされにくくすることができます。

---

## ■ラスタンサーガ

**・タイトー：1987：**AC、PS2、PSP

**・剣**

ファンタジー世界を題材にしたゲーム。主人公の戦士を操作して、数々の敵を倒しつつ、ボスのドラゴンを目指します。アイテムを拾うことによって、剣・斧・鉄球といった武器を切り替えて使うことができます。レバー入力やジャンプボタンと攻撃を組み合わせることによって、しゃがみ斬りやジャンプ斬り、下突きといった攻撃も可能です。

---

## ■ラッシュ&クラッシュ

・カプコン：1986：AC

・動物（自動車）

主人公を操作して、敵の攻撃をくぐり抜け、悪者にさらわれた家族を助けにいくゲーム。内容としては「フロントライン」に近いゲームです。最初に主人公は車に乗って登場します。車は一定のダメージを受けると炎上するので、爆発する前に脱出する必要があります。脱出した後は生身で進むことになりますが、車に比べると移動速度が遅く、攻撃も弱く、さらに一発でも攻撃が当たるとミスになってしまいます。車のときと生身のときで、操作方法がまったく違うのが面白いところです。

---

## ■ラリーX（ラリーエックス）

・ナムコ：1980：PS、PSP、PS2、XB360

・煙幕、すべてのアイテムを取る

上下左右にスクロールする迷路状のステージで、自分の車を操作し、迫ってくる敵の車を避けながら、ステージに配置された旗を回収するゲーム。ボタンを押すと車から煙幕を出すことができ、煙幕に巻き込まれた敵の車は一定時間動けなくなります。煙幕を使うたびに燃料が減っていき、燃料がなくなると車のスピードが落ちてしまいます。ほぼ同内容の作品に、難易度や音楽などを変更した「ニューラリーX」があります。

---

## ■龍虎の拳（りゅうこのけん）

・SNK：1992：AC、NG

・三角跳び

素手で闘う格闘ゲーム。基本的なゲーム内容は「ストリートファイター」シリーズを踏襲していますが、大きなキャラクターのグラフィックや、拡大縮小機能を使ったズームイン・ズームアウトの演出など、独自性も見られます。必殺技に加えて、より強力な超必殺技があり、複雑なコマンド入力を要求されます。また、超必殺技を出すために気力をためるアクションも独特です。

---

## ■ルナーランダー

・アタリ：1979：AC

・目的地へいく

着陸船を操作して、月に着陸するゲーム。慣性や重力が働く独特の操作感覚と、ワイヤーフレームで描かれたグラフィックが印象的です。着陸船のコントロールは、エンジンの推力を調整するレバーと、着陸船を回転させる姿勢制御ボタンで行います。開始時に難易度を選ぶことができ、難易度によって重力の強さや、大気抵抗の有無が選べます。

---

## ■レインボーアイランド

・タイトー：1987：AC、DS、FC、MD、PS2、PSP

・足場を作る

足場を作りながらステージを登っていくゲーム。ゲーム内容はまったく違いますが、ストーリー上では「バブルボブル」の続編に相当するゲームです。前作とは違い、2人同時プレイはありません。主人公はボタンで虹をかけることができます。虹を使って敵を倒したり、虹を足場にしてジャンプしたりしながら、ステージを登っていくことが目的です。虹の上に登ると虹を落とすことができ、落とした虹に敵を巻き込んで倒すこともできます。

---

## ■レッキングクルー

・任天堂：1984：AC、FC、GBA、SFC

・すべてのアイテムを取る

主人公を操作して、ビルの解体作業を行うゲーム。敵から逃げながら、ハンマーで壁を叩いて壊します。すべての壁を壊すとステージクリアです。ステージ内に置かれた爆弾を使うと、壁をまとめて壊すことができます。ファミコン版にはステージのエディット機能があります。アーケード版では2人同時プレイが楽しめます。

## ■レディバグ

・ユニバーサル：1981：AC

・回転ドア

主人公のてんとう虫を操作して、ステージに配置されたエサを集めるゲーム。「パックマン」に似たドットイーターゲームですが、独自の要素として、回転ドアが導入されています。主人公が回転ドアを押すと、ドアが回転します。ドアを上手に回転させ、通路の形を変えることによって、敵の追撃をかわすことが攻略のポイントです。

## ■ロードランナー

・ブローダーバンド：1983：AC、DS、FC、GBA、PC、PS、SFC、SS

・ロープ、はしご、自動穴、すべてのアイテムを取る

ステージ内に配置された金塊を回収するゲーム。主人公は左右の床に穴を掘ることができ、穴に敵を落として足止めしたり、敵を埋めて倒したりすることができます。また、穴を使って飛び降りたり、穴に入ってさらに穴を掘ることで、地中深くに潜ったりすることも可能です。ブローダーバンド以外に、各社がコンシューマ版やアーケード版を発売しています。例えばファミコン版はハドソン、アーケード版はアイレムから出ています。

## ■ロックンロープ

・コナミ：1983：AC、DS、PS

・ロープを張る

主人公の登山家を操作して、ロープを駆使しつつ、山の頂上にいるジュジャクと呼ばれる鳥のところまで登るゲーム。ボタンを押すとロープが出て、ロープが壁などに当たると張ることができます。主人公はロープをつたって上の階に登ることが可能ですが、敵のなかにはロープをつたって主人公に迫ってくるものもあるので、注意が必要です。敵を懐中電灯で照らして気絶させ、一定時間行動不能にさせることもできます。

## ■ロンパーズ

・ナムコ：1989：AC、PS

・壁を倒す

迷路のようなステージを歩き回り、壁を使って敵を倒すゲーム。壁は押すと倒すことができ、敵を壁の下敷きにすれば退治することが可能です。壁には幅の広いものもあるため、これを利用して複数の敵を同時に退治すると高得点です。ステージ内に配置された鍵をすべて回収すれば、ステージクリアとなります。

## ■ワープ&ワープ

・ナムコ：1981：AC、FC

・時限爆弾

主人公を操作して、迫りくる敵を倒すゲーム。敵をすべて倒すとステージクリアとなります。ステージには空間の面と迷路の面の2種類があり、前者では銃を使って、後者では爆弾を使って敵を倒します。爆弾は時限爆弾で、爆発すると四方に爆風が広がります。この爆弾の動きは「ボンバーマン」と同じですが、「ワープ&ワープ」の方が先行する作品です。シューティングゲームとアクションゲームがミックスされたような、ちょっと不思議な作品です。

## ■ワルキューレの伝説（ワルキューレのでんせつ）

・ナムコ：1989：AC、PS

・複数キャラクターの操作（複数人数プレイ）

主人公の少女ワルキューレを操作して、数々の敵を倒しながら、ステージを進んでいくゲーム。ファミコン版の「ワルキューレの冒険」の続編に相当する作品ですが、グラフィックやサウンドが大幅にパワーアップしており、まったく別の作品という印象です。2人同時プレイが可能で、プレイヤー1がワルキューレを、プレイヤー2がサンドラと呼ばれるキャラクターを操作し、協力プレイが楽しめます。

## ■ワンダーボーイ

・ウエストン/セガ：1986：AC、PS2

・スケートボード

横スクロールのアクションゲーム。主人公の少年を操作して、敵や障害物を切り抜けつつ、ステージを進んでいきます。ステージに配置されているスケートボードに乗ると、速いスピードで移動することができます。また、敵に接触してもスケートボードから降ろされるだけで、ミスにはなりません。

Appendix 3

# 索引

## ■英数字



## ■あ行



## ■か行

■さ行



■た行

■な行



■は行



■ま行



■や行

■ら行



■わ行



■サンプル

### ■引用ゲーム

■クラス

# おわりに

「アクションゲームを遊ぶ人は、一度くらい『自分でアクションゲームを作ろう』と思ったことがあるはず！」という考えのもとに、普通のゲームプログラミング入門書とは違った切り口で進めてきた本書でしたが、お楽しみいただけたでしょうか。本書がアクションゲームに関してあれこれ考えたり、楽しく遊んだり、熱く語ったり、プログラムを書いたり、いっそうアクションゲームが好きになったりするための手助けになれば幸いです。

最後に、本書の関連書籍を紹介させていただきます。本書とあわせてお読みいただけば、ゲームプログラミングの世界をより深く楽しんでいただく助けになるかと存じます。ぜひお試しください。

**『シューティングゲームアルゴリズムマニアックス』**

古今東西のさまざまなシューティングゲームにおけるギミックを解説した、本書の姉妹書籍です。本書と同じく図解を中心にしているので、漫画を読むような気楽な気持ちで、シューティングゲームの世界を堪能していただけます。

ISBN4-7973-2731-6　定価：2,940円（本体＋税）　B5変形版

**『ゲームエフェクトマニアックス』**

数々の3Dエフェクトを作成する方法を満載した書籍です。3Dグラフィックスやシェーダープログラミングに興味をお持ちでしたら、カッコいいゲームを作るためにきっと役立てていただけるでしょう。

ISBN4-7973-3295-6　定価：2,940円（本体＋税）　B5変形版

**『シューティングゲームプログラミング』**

シューティングゲームの制作方法を実践的かつ平易に解説した書籍です。これからシューティングゲームを作ろうという方にはもちろん、シューティングゲームを題材にプログラミングを学ぼうという方にもお使いいただけます。ゲームライブラリやタスクシステムに関する詳細な解説もあります。

ISBN4-7973-3721-4　定価：2,940円（本体＋税）　B5変形版

■著者プロフィール

**松浦 健一郎**（まつうら けんいちろう）
東京大学工学系研究科電子工学専攻修士課程を修了後、研究所勤務を経て、現在は趣味と実益を兼ねつつフリーのプログラマ＆ライターとして活動中。著書（いずれも司ゆきと共著）に「はじめてのJBuilder4」「Delphi DB&Webプログラミング」「はじめてのJBuilder6」「デスクトップマスコットを作ろう!!」「シューティングゲーム アルゴリズム マニアックス」「ゲームエフェクト マニアックス」「Javaのココロ」「シューティングゲーム プログラミング」（以上、ソフトバンククリエイティブ刊）、「Windows Vistaガジェットプログラミング」（秀和システム刊）がある。関心と仕事の範囲はプログラミングを中心にコンピュータ全般に及ぶが、最も興味がある分野はプログラミング言語作りとゲーム作り。

**司 ゆき**（つかさ ゆき）
東京大学理学系研究科情報科学専攻修士課程修了。プログラマおよびライター。最近の成果はシイタケ栽培のための地下室を獲得したこと。

著者Webサイト「ひぐぺん工房」
http://cgi32.plala.or.jp/higpen/gate.shtml

## アクションゲーム アルゴリズム マニアックス

2007年5月30日　初版第1刷発行

著　者・・・・・・・・・・松浦 健一郎（まつうら けんいちろう）／司（つかさ）ゆき

発行者・・・・・・・・・・新田 光敏
発行所・・・・・・・・・・ソフトバンククリエイティブ株式会社
〒107-0052　東京都港区赤坂4-13-13
TEL 03-5549-1201（販売）
http://www.sbcr.jp

印　刷・・・・・・・・・・株式会社シナノ
装丁/本文デザイン/組版・クニメディア株式会社

落丁本、乱丁本は小社販売局にてお取り替えいたします。
定価はカバーに記載されております。

Printed In Japan　　ISBN978-4-7973-3895-9

SoftBank Creative

ISBN978-4-7973-3895-9
C0055 ¥2800E
9784797338959
定価 本体2,800円 +税
1920055028004

アクションゲーム アルゴリズム マニアックス

ActionGame Algorithm Maniax

松浦健一郎／司 ゆき 著

SoftBank Creative